MOOIDE: Natural Language Interface for Programming MOO Environments

by
Moinuddin Ahmad

Bachelor of Technology
Indian Institute of Technology

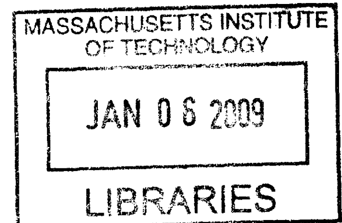
Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008



© Massachusetts Institute of Technology 2008. All rights reserved.

Author ______________________________
Program in Media Arts and Sciences
September, 2008

Certified by ______________________________
Dr. Henry Lieberman
Research Scientist
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by ______________________________
Prof. Deb Roy
Chairperson, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

ARCHIVES

MOOIDE: Natural Language Interface for Programming MOO Environments

by
Moinuddin Ahmad

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on September, 2008, in partial fulfillment
of the requirements for the degree of
Master of Science

Abstract

MOOIDE is an interface to allow novice users to program a MOO environment using natural language. Programming the MOO involves a variety of tasks like creating objects and their states, assigning verb actions to objects, and programming behavior that changes states of objects and generates messages. Once the MOO is programmed, other users can interact with the objects for entertainment or educational purpose.

To make MOO programming easier and more accessible to novice programmers, our natural language interface allows users to describe different MOO programming tasks in English. These include adding objects, object properties, states and relationships between objects. They also include verbs through which behaviors are accessed in the MOO. Users can use English to describe decision statements, loops, conditions and other typical programming constructs.

Earlier systems focused on addressing parsing issues in programming. However, those systems lacked commonsense knowledge. MOOIDE brings commonsense features to natural language programming in addition to parsing. Commonsense reasoning allows MOOIDE to automatically include typical object properties, verb affordances and affordance rules as well as typical verb effects. Such augmentation of natural language programming with commonsense reasoning capabilities can help make programming significantly more intuitive to novice programmers.

Thesis Supervisor: Dr. Henry Lieberman
Research Scientist
Program in Media Arts and Sciences

MOOIDE: Natural Language Interface for Programming MOO Environments

Moinuddin Ahmad

Thesis Reader ____

Prof. Mitchel Resnick
Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Massachusetts Institute of Technology

MOOIDE: Natural Language Interface for Programming MOO Environments

Moinuddin Ahmad

Thesis Reader ____

Prof. Nick Montfort
Assistant Professor of Digital Media,
Program in Writing and Humanistic Studies,
Massachusetts Institute of Technology

Acknowledgment

Foremost, I would like to thank my advisor Henry Lieberman who encouraged me to take up this work in the first place and kept me focused and on track. He was a source of continuous intellectual input and feedback.

My readers Nick Montfort and Mitch Resnick were a source of significant intellectual input in this work. Nick, a graduate from the Media Lab is quite a broad thinker and his questions always got me to think about areas that I had not thought about.

Arriving at the Media Lab, I was significantly influenced by Marvin Minsky, his work and his ideas elaborated in The Society of Mind and The Emotion Machine. More importantly, his commitment to the path towards strong AI—a path that many have abandoned. His impriming will remain strong and keep me motivated to keep working on AI in the future.

The Commonsense Computing Group at the Media Lab - Henry, Dustin, Ian, Ken, Edward, Rob, Bo, Jason, Mako, Barbara, Catherine, Vina and many other folks who participate in our group. I have significantly developed intellectually in the company and influence of this group.

I never met Push Singh. But his work on computational models for reflective thinking was highly inspirational as were a lot of his other writings. In part this work was inspired by the idea to acquire reflective procedures through people using natural language.

Linda Peterson is quite an able administrator, aptly balancing her task to get students to finish within the dates and deadlines while avoiding undue pressure that could negatively influence the quality of creative work.

And finally my friends and family without who's support and influence I wouldn't have been at MIT in the first place.

Its MOOIDE...MOO-DEE, if you will.

Table of Contents

CHAPTER 1 .... **9**
Introduction 9
1.1 Programming in Natural Language 9
1.2 MOO Environments 11
1.3 MOO Programming 13
1.4 MOOIDE Architecture 14
1.5 A sample scenario of MOOIDE user interaction 16

CHAPTER 2 .... **21**
MOOIDE System Description 21
2.1 Overview 21
2.2 The MOOIDE User Interface 23
2.3 The Parsing System 26
2.3.1 Object creation, properties, states and relationships 28
2.3.2 Verb definitions 29
2.3.3 Verb argument rules 30
2.3.4 Verb program generation 32
2.3.5 Decision constructs 33
2.3.6 Decision conditions 35
2.3.7 Verb commands 35
2.3.8 Iterators, variables and loops 37
2.3.9 Messages 37
2.3.10 Anaphora resolution 38
2.4 Commonsense features 39
2.4.1 Learning commonsense knowledge 40
2.4.2 Suggesting verbs 41
2.4.3 Universal verbs and properties and cause-effect rules 41
2.5 Dialogs and messages 42

CHAPTER 3 .... **43**
Evaluation 43
3.1 Experiment hypothesis 43
3.1 Experiment protocol 43
3.2 Familiarization scenario 44
3.3 Experiment task scenario 44
3.4 Post test questionnaire 45
3.5 Study results 45
3.6 Observations 46

CHAPTER 4 .... **49**
Discussion 49
4.1 Commonsense in MOO environments 49
4.2 Learning programming 51
4.3 Understanding natural language 51
4.4 Natural language programming for formal languages 53

CHAPTER 5 .... **56**
Related Work 56

APPENDIX **60**
Appendix – I : Grammars and Parse Frame Definitions 60
Appendix – II : Evaluation Sequence 65

BIBLIOGRAPHY **68**

List of Figures

Figure 1 : A typical Zork session....12
Figure 2: Example of MOO code....13
Figure 3: MOOIDE Interface....15
Figure 4: MOOIDE Conversation Window....23
Figure 5: MOOIDE Simulation Window....24
Figure 6: MOOIDE Program Description Window....25
Figure 7: MOOIDE Commonsense Facts Window....25
Figure 8: MOOIDE Load Facts Window....25
Figure 9 : Post Evaluation Questionnaire Summary....45

Chapter 1

Introduction

1.1 Programming in Natural Language

Why would we want to program using natural language ? Have we humans not developed syntactically precise programming languages with well-defined semantics for the purpose of avoiding the inherent ambiguity in language ? The answer to this question is that yes we did, and it led to the vast science of computer science which is focused on describing procedural information. The software industry has also developed vast implementation expertise by undertaking projects that go from simple initial specifications to complicated programmed behaviors. Now it is time to make use of this expertise to make programming easier and more accessible to people who do not know or do not have the time to learn precise syntax and semantics of common programming languages.

The task is not simple or straight forward. Solving completely programming in natural language would probably be an 'AI complete' problem to solve and would require the natural language programming system to parse and understand a very large variety of interactive programming situations. However, humans are adaptable to constrained interfaces as long as the interfaces are intuitive to some extent. For example, humans can quickly understand syntactic nuances of natural language interfaces. Natural language is certainly easier to write than programming language code because humans are quite familiar with its general syntax. However, allowing people to use natural language will create the problem of underspecification which will be frustrating to users. Users using common programming languages for programming are accustomed to the idea of detailed procedural specification. Natural language, on the other hand, allows its users to have significant amount of ambiguity and underspecification. People

communicating procedures, therefore, rely on commonsense to interpret and complete underspecified procedures. Underspecification comes across in many different ways—procedural (incomplete list of instructions), descriptional (lack of commonsense facts about objects), of constraints (limitations on behavior) etc. User input often has have ambiguities and contradictions. Each of these problems will need to be handled by the natural language interpreter. However, by using different methodologies like dialog modeling and commonsense reasoning, we can attack these problems.

In this work, we have explored ways in which a novice programmer might use natural language to describe and program a common programmable environment—the MOO/MUD environment (short for Multi User Object Oriented/Multi User Dungeons and Dragons [1]). Certain earlier work like Metafor [2] and Inform7 [3] has focused on parsing to describe programming constructs in natural language. We have explored ways in which a programmer's life could be made easier by the addition of commonsense knowledge [26] to the environment.

There are numerous projects around the world which focus on collecting and representing commonsense knowledge. One of the earliest project, the Cyc [25] project uses logical rules to represent commonsense knowledge. For example the fact "Bill Clinton is a President" is represented as `(#$isa #$BillClinton #$UnitedStatesPresident)`. Other projects like the OpenMind [4] project collects facts as English statements like "People generally sleep at night" that are acquired from people. With a large database of commonsense rules, a user would require significantly less programming effort. Commonsense rules used in a programmed environment should be in simple English syntax so that they can be acquired by users through an interface like OpenMind and shared between applications using a commonsense reasoning system like ConceptNet [5].

Virtual reality environments like the MOO environments often have their own rules and behavior that overrides daily commonsense—for example in a MOO world one might have a congenial dragon who hugs you when you greet him. In such a world you would not get burned

trying to kiss the dragon. We provide a methodology through which commonsense rules like "When you touch fire, you get burnt." can be overridden in the programmed virtual reality and packaged to build a new world e.g. dragon world.

Giving a natural language interface to programming in an environment requires an intuitive model of programming pragmatics and an understanding of the ways in which a typical programmer would think about that environment.

1.2 MOO Environments

A MOO is a virtual reality environment that enables people to interact with the objects in the environment in a variety of ways through the use of text. Users log into the system and initiate actions with any of a variety of typed commands and read the results as text displayed on the screen. Within the virtual reality environment, MOO worlds can have a quite rich representation of a variety of real and fictional scenarios. MOO worlds can take the shape of adventure games like Zork [22], earthly environments like LambdaMOO [1] and environments for educational purposes like schMOOze [24] and MediaMOO [23]. Commonly in a MOO scenario, there are many places called 'rooms' which are connected by 'exits'. One can explore the scenario using commands such as "go north", "go south", "enter cottage" etc. Interactive objects are also available in the scenario and are presented by their textual descriptions such as "A surprise waits you in the magic box", and these objects can be manipulated using commands such as "open box", "look in box" and "take surprise prize from box". When a player types one of these verb commands, appropriate text describing the effect of such a command will appear on the screen and the environment will ensure that the system does the appropriate state changes. The core part of programming MOOs is programming the effects of these verb commands.

Below is a transcript of a session of the interactive fiction game Zork. Zork was one of the earliest games based in a text based virtual reality. Note text post the character ">" are the

commands that the user playing the game issues into the virtual reality. Also note the messages that the game generates when the player types different commands.

```
ZORK I: The Great Underground empire
Copyright (c) 1981, 1982, 1983 Infocom, Inc. All rights reserved.
ZORK is a registered trademark of Infocom, Inc.
Revision 88 / Serial number 840726

West of House
You are standing in an open field west of a white house, with a boarded
front door.
There is a small mailbox here.

>open mailbox
Opening the mailbox reveals a leaflet.

>read leaflet
(taken)
"WELCOME TO ZORK!
ZORK is a game of adventure, danger, and low cunning. In it you will
explore some of the most amazing territory ever seen by mortals. No
computer should be without one!"

>go down
The trap door crashes shut, and you hear someone barring it.

Cellar
You are in a dark and damp cellar with a narrow passageway leading north,
and a crawlway to the south. On the west is the bottom of a steep metal
ramp which is unclimbable.
Your sword is glowing with a faint blue glow.

>go north

The Troll Room
This is a small room with passages to the east and south and a forbidding
hole leading west. Bloodstains and deep scratches (perhaps made by an
axe) mar the walls.
A nasty-looking troll, brandishing a bloody axe, blocks all passages out
of the room.

Your sword has begun to glow very brightly.
The troll swings his axe, but it misses.

>swing sword
Whoosh!
The troll swings, you parry, but the force of his blow knocks your sword
away.

>get sword
Taken.

The troll hits you with a glancing blow, and you are momentarily stunned.

>kill troll with sword
The troll is staggered, and drops to his knees.
The troll slowly regains his feet.
```

Figure 1 : A typical Zork session

1.3 MOO Programming

Different MOO environments have different programming languages in which they can be programmed. There is a standard programming language called the MOO programming language (a dynamically typed prototype based object oriented language, syntactically somewhat derived from Algol) which is popular among MOOs, however, there are MOOs (e.g. MOOP) that use a language like Python to enable programmability. Python gives enormous flexibility to a programmer for procedural expression and is also a much more general purpose language than the MOO programming language. For this reason we generate Python code instead of the common MOO code.

In the MOOs, the terms program, command and verb are often used interchangeably because behaviors are embodied in the verbs. Every verb is attached to an object. The server parses a line of text and tries to identify the object on which the verb should execute. The first word of the command is always the name of the verb and the rest are arguments. If it can't identify an object with an appropriate verb to run on, it generates an error message.

Programming the MOO environment is typically a fairly constrained tasks. Unlike general purpose programmable environments like MATLAB or the Unix Shell in which programmers can make elaborate algorithms, users exploring the MOO programming environments have considerably simpler expectation of how verbs behave.

```
player:tell("You take a cookie from the jar.");
player.location:announce(player.name, " takes a cookie from
            the cookie jar and eyes it hungrily.");
if (player.name == "Cookie Monster")
     player:tell("You wolf down the cookie with gusto.");
     player.location:announce(player.name, " gobbles up the
            cookie with no regard for decorum.  You are
            splattered with crumbs.");
else
     player:tell("You delicately nibble at the cookie.");
     player.location:announce(player.name, " demurely

            nibbles at the cookie and politely cleans up
            the crumbs.");
endif
```

Figure 2: Example of MOO code

Behavior in MOO consists of text based responses to 'verb-actions' that users invoke on objects. For example let us say the user creates an object 'cookie', and a verb 'gobble'. When somebody invokes the verb by entering 'gobble cookie' into the MOO prompt, the system could print out "You gobble the cookie with gusto!" So, as long as the MOO system gives an interesting response in the MOO text window and maintains some amount of state, for instance making sure the cookie is no longer available to be gobbled or eaten, it would be interesting enough. Some MOOs have long story lines, these are built by having a variety of rooms and objects rather than by building objects with extremely complicated behaviors.

Since the MOO is a text based environment, the application is also not expected to do complicated spatial movements or understand in depth how real world objects behave. Any sort of state management in the MOO needs to be programmed into it by the programmer. In our system, we take that as a commonsense problem and address it by using a verb database containing default behaviors, for example cause-effect rules like "When you drop a delicate object, it breaks". Users can also add such commonsense knowledge that is not available in the database.

1.4 MOOIDE Architecture

To enable natural language programming in the MOO environment, MOOIDE uses the approach of a desktop application (a web based interface would also work especially if the system needs to be deployed to a large number of users).

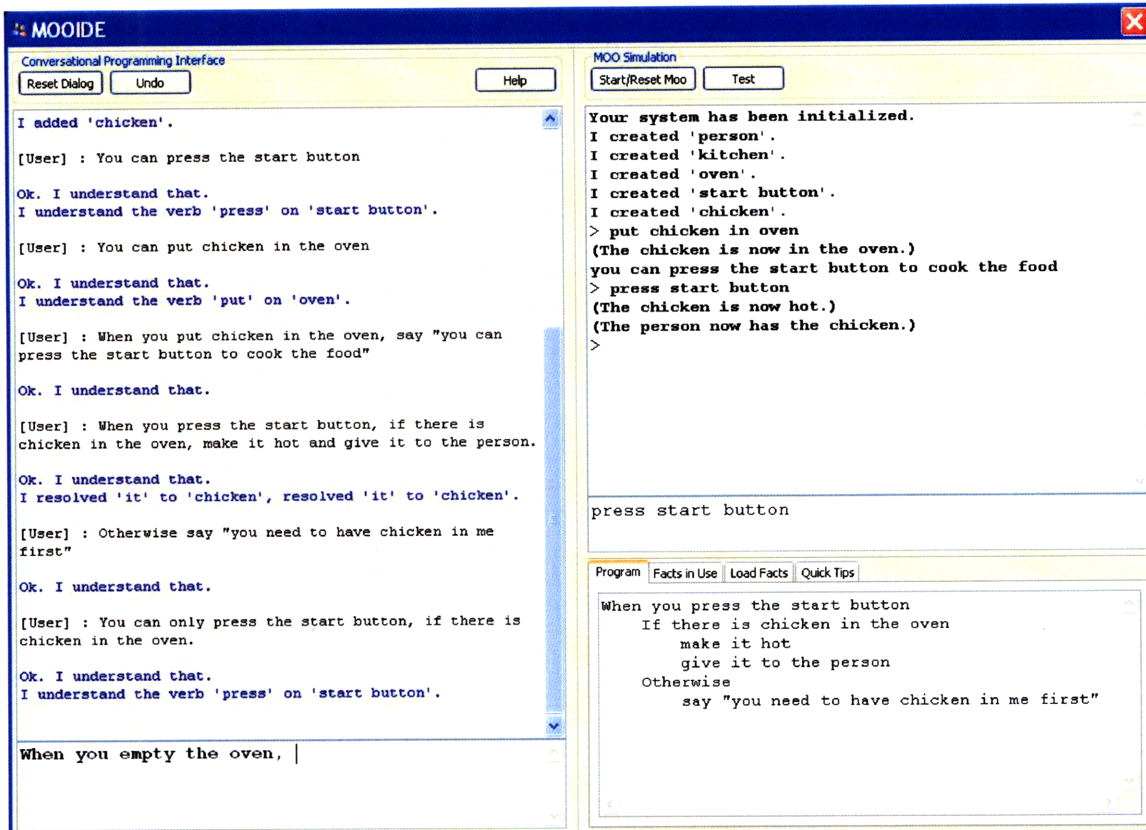


Figure 3: MOOIDE Interface

Users interact with the natural language dialog system on the left window and they can test the described world in a MOO simulation inside the window on the upper right. The window on the lower right gives a summary of the programmed verbs in English as well as commonsense rules that apply to those verbs.

User input is parsed by different specialized parsers for different parts of the input. We used chunking over parse trees and frame based parsing for different purposes. Depending upon the context, the parser resolves any anaphoric references to objects previously created by the user, generic commonsense objects or program arguments and variables. Then it generates code and adds it to a hierarchical internal code representation. When the user activates the MOO simulation for testing, the system generates Python code and pushes it into the MOO simulator where the user can test and explore the environment he created.

User input can be either descriptional—objects in the environment, their properties and relationships between them or procedural—state changes, output messages, affordances etc. Depending upon the context, the system tries to guess if the input is a commonsense fact and if it is overriding earlier facts. It stores commonsense facts separately until they can be vetted before being entered into the main database. The user can also save the world knowledge separately so that it can be loaded later.

1.5 A sample scenario of MOOIDE user interaction

To get a sense of the capabilities of MOOIDE, let us look at some representative examples of the interactions MOOIDE handles. These examples are situated in a common household kitchen where a user is trying to build objects and giving them behaviors. Let us look at some inputs that he put into MOOIDE:

```
There is a chicken in the kitchen.
There is an oven.
You can only cook foods 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, chicken and oven and gives the "cook" verb to the oven. 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 foods 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 'chickens are types of foods' the MOO generates an error—"You can only cook food in oven." This is not what the user expected. To resolve this error, he then has to add the statement "Chickens are types of foods". 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."

Let us look at another example similar to the previous example :

```
There is a cat in the kitchen.
Cats are animals.
There is a garbage box in the kitchen.
You cannot throw animals in the garbage
box.

When you throw things in the garbage box,
if the garbage box is full, say "Sorry we
are full today".

Otherwise say "The garbage box now has
these items" followed by the contents of
the garbage box.

If there is an item in the garbage box, the
garbage box becomes full.
```

Thanks to our English syntax, the "code" above is quite self explanatory. However, notice that nowhere in the description, the user has specified that after 'throwing' something into the garbage box the relationship `in(object,garbage box)` actually becomes valid. This relationship can also be referenced by the `has(object,garbage box)` relationship. MOOIDE makes a commonsense assumption that this relationship will be valid after the `throw_in` verb. Furthermore it parses "contents of the garbage box" to the function `contents_of(garbage box)` that returns all the objects with the 'in' relationship. Let us continue.

The user adds more to the above example:

```
The garbage box has a push button on it.

When somebody presses the button, empty the
garbage box.
```

Now, note that in the first example, the user refers to a property change by the command "make it hot". Where "make" becomes the function and "it" and "hot" are the variables. In this case, the verb command "empty the garbage box" is referenced before it has been defined. MOOIDE does not know how to handle this verb. So it asks the user:

"I do not know how to empty the garbage box. You can help me with that. How do you empty a garbage box?"

The user replies:

```
To empty the garbage box, take its contents
and put them on the floor.
```

Note that this definition identifies iterators and creates temporary variables—both for contents of the box. After this definition, the user wishes to go back to editing the original code and not the definition of the 'empty' verb. He says:

```
After emptying the garbage box, say "The
garbage box is now empty. You can put more
things in it."
```

The "after" statement allows MOOIDE to switch the programming context to the earlier behavior description. However, note one issue, the system does not know that Empty and Full are mutually exclusive properties (or are at least related in some way). In the example above, the logic of the system does not fail because of the lack of knowledge of this fact. The user may add:

```
When you try to empty the garbage box, if
it is already empty, say "The garbage box
is already empty."
```

Or he might add:

```
When somebody presses the push button, if
the garbage box is empty, say "The garbage
box is already empty."
```

In both these cases, MOOIDE understands that the property 'empty' of the garbage box is related to the verb 'empty the garbage box' and it automatically assigns this property to it. However, this was possible only because the user created a new verb "empty garbage box". If he had not create this verb, the logic would have failed because MOOIDE would not know how to relate the "empty" and "full" properties.

Note also the "When you try" construct. This statement is the definition of exception handling and it will be processed when it is expected that the verb will fail to execute (maybe because of a verb argument rule).

This leads us into another example similar to the above in which MOOIDE handles normality properties with object hierarchies. MOOIDE has a way to load standard commonsense knowledge. The user starts with the following pre-packaged commonsense facts.

```
Containers can be full or empty.
They are normally empty.
Garbage boxes are containers.
You can only put things in a container,
if it is empty.
Cats are animals.
You cannot throw animals in a garbage box
```

Then he adds objects and gives some behavior:

```
There is a cat in the kitchen.
There is a tomato in the kitchen.
There is a potato in the kitchen
When you throw something in the garbage box,
the garbage box becomes full.
```

This particular example is a way to build the previous example through commonsense rules. Should there be a large database of commonsense facts, it would be very easy to build objects and behaviors through MOOIDE. Note in this example, the properties "full" and "empty" are mutually exclusive. The system can also handle different syntactic varieties of the statements above for example one might say “containers may be full or empty” or “containers are usually

empty. Let us say the user tries to throw two thing in the garbage box—one after the other. The interaction would be something like this:

```
[user] >    throw potato in garbage box
[system]    The garbage box is now full.
[user] >    throw tomato in garbage box
[system]    Sorry you cannot do that
            because you can only
            put things in a container,
            if it is empty.
```

As we see in the above case, MOOIDE handles properties through its hierarchy of objects and applies them to behavior rules that apply. Users can also refer to properties of parent objects without explicitly asking them to be inherited.

One of the issues with earlier systems—the user/programmer has to enter all commonsense knowledge like the above statement by himself. So in this system, we have tried to identify different issues around integrating commonsense knowledge and commonsense reasoning with programming in natural language. MOOIDE also automatically identifies that from all the input above, the statement "you can only cook food in an oven" is a commonsense fact and saves it for later use. As the commonsense database grows, users will be able to build behaviors more easily, accurately and intuitively.

Chapter 2

MOOIDE System Description

2.1 Overview

In this section we will describe the basic working of our system. MOOIDE as previously described, is an interface to allow users to program behavior in natural language. This involves many different things—parsing, program generation, commonsense reasoning etc. 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 from that generates parse trees by utilizing a grammar (often a probabilistic context free grammar). For example the sentence "You can put water in a bucket." 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]
```

In this case, this frame pattern matches "water" to "ARG" and "a bucket" to "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 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.

2.2 The MOOIDE User Interface

As described previously, the MOOIDE interface consists of three different windows.

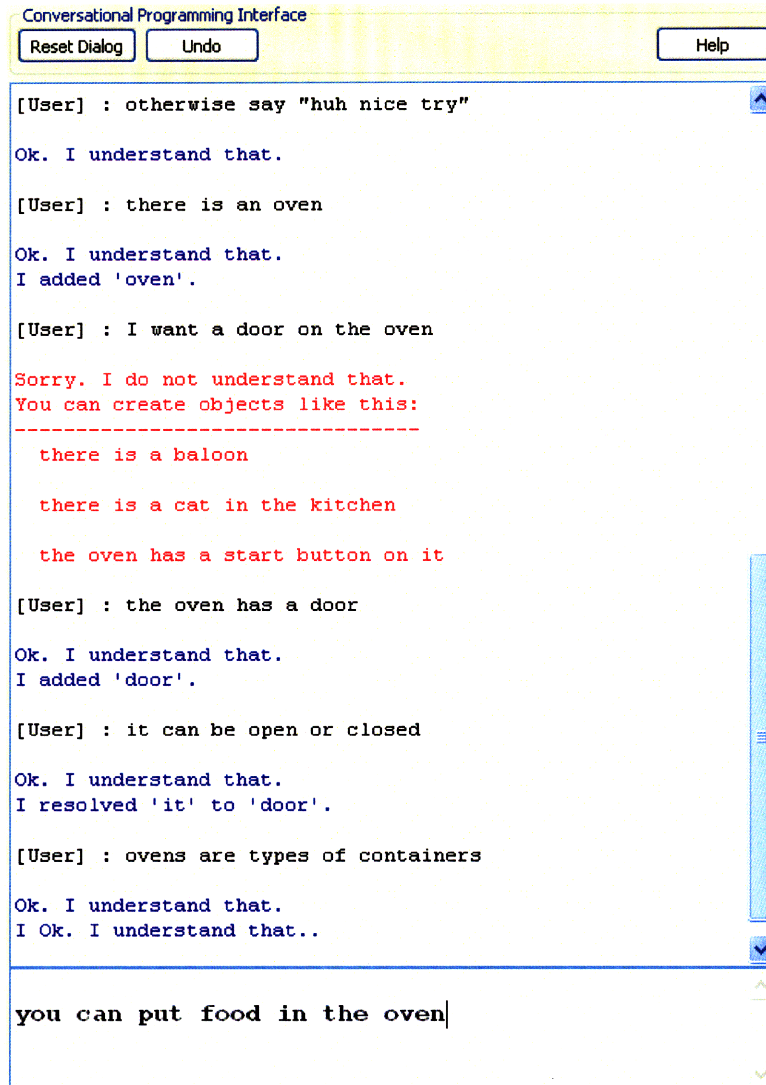


Figure 4: MOOIDE Conversation Window

The left window is the main conversation window in which the user can type English descriptions of the code that he has in mind. Responses from MOOIDE are also printed out in this window. The interface has a Reset and an Undo button. The Reset button completely resets the program description that the user provided and removes any test code as well. It also resets the

commonsense facts being used. Commonsense facts must be reloaded if they are required again. MOOIDE does not have extensive capabilities to allow changes to previous descriptions so the the user needs to use the Undo button to correct errors.

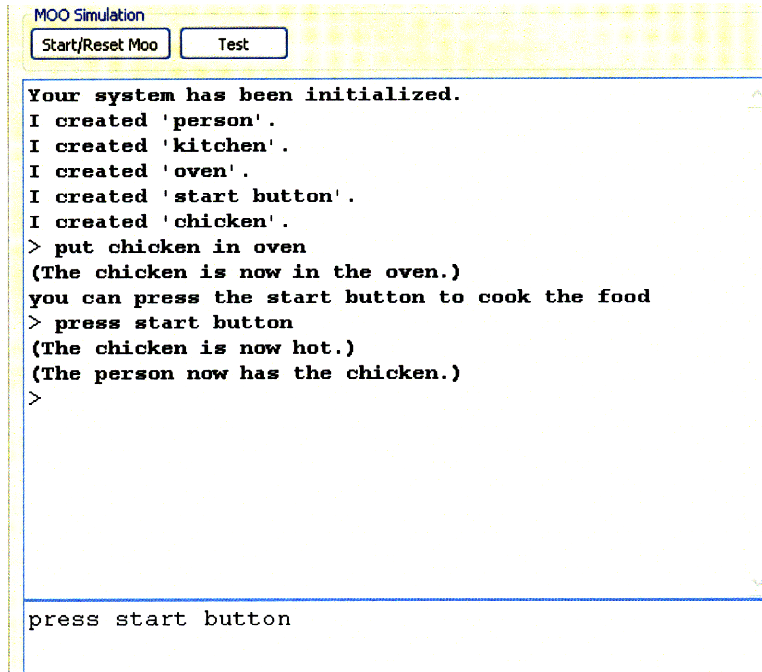


Figure 5: MOOIDE Simulation Window

The top-right window has a MOO simulator in which a user can enter standard MOO commands. The MOO simulator uses an integrated MOOP server. The MOOP (MOO/Python) system is a 3rd party MOO server written in Python. Python is also the core programming language for the verbs written in MOOP. The simulator has 'Run/Reset MOO' and 'Test' buttons. The 'Run/Reset MOO' starts a blank MOO system with no objects. Users can use standard MOO programming to build up the blank MOO system. The 'Test' button converts the current program representation to code and loads it into the MOO simulator so that the user can test if the generated code matched the requirements he had in mind.

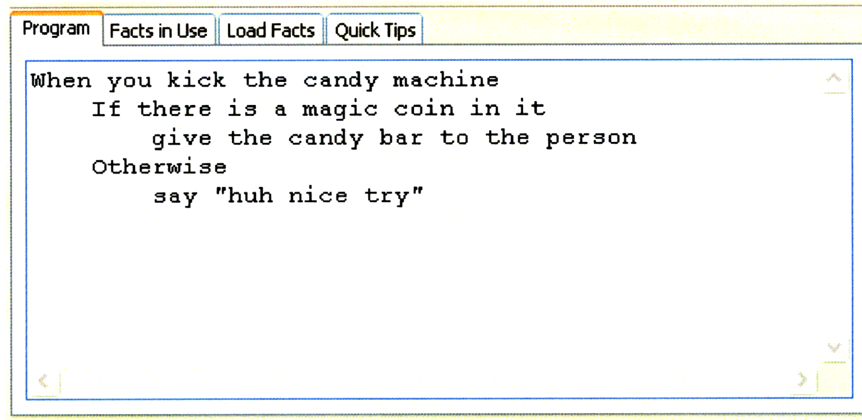


Figure 6: MOOIDE Program Description Window

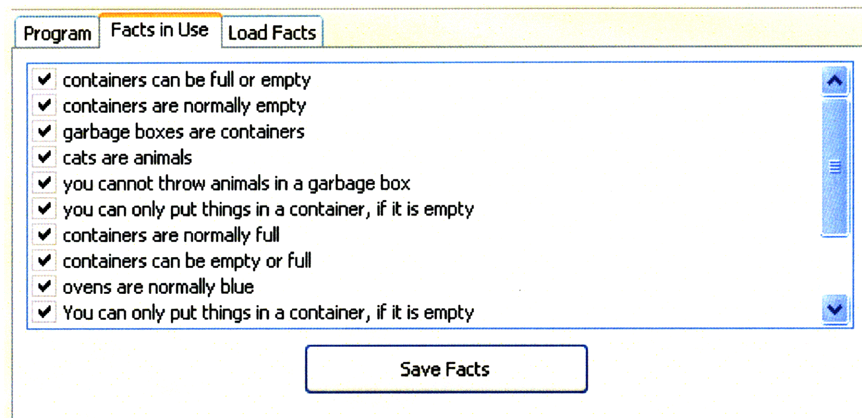


Figure 7: MOOIDE Commonsense Facts Window

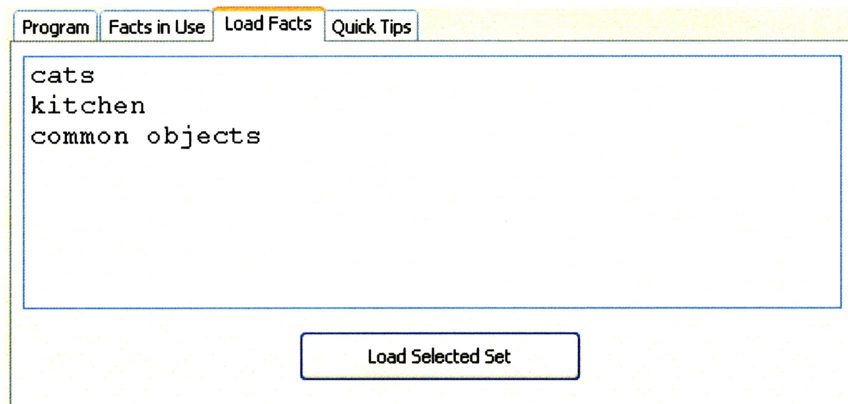


Figure 8: MOOIDE Load Facts Window

The bottom-right window has four tabs. The first tab is titled "Program Description". The second tab "Facts in use", the third tab is "Load facts" and the fourth tab is “Quick Tips”. The

"Program Description" tab contains an English description of the particular verb that is being described. Basically the parser structures and formats the English statements that the user provided so that he can get a coherent view of the code as it is interpreted by the parser.

The "Facts in use" tab provides a list of commonsense facts have been loaded into the environment as well as any other facts that were automatically identified in the user's descriptions. The user can save a set of facts into a new or an existing "Collection". Before doing that, he can select which facts he wants to save (because certain facts might not have been accurately acquired). In the "Load facts" tab, the user can load a collection into the current environment.

The Quick tips tab provides with basic in-context help. For example if a user gets an error while trying to describe a When clause, he will be given a few examples of that clause.

2.3 The Parsing System

The parser implementation uses the parser developed by Stanford Natural Language Processing Group. The Stanford parser is a Probabilistic parser that uses knowledge of language gained from hand-parsed sentences to produce the most likely analysis of new sentences. Statistical parsers still make some mistakes, but commonly work rather well. Their development was one of the biggest breakthroughs in natural language processing in the 1990s. The parser is a Java implementation of probabilistic natural language parsers, both highly optimized PCFG and lexicalized dependency parsers, and a lexicalized PCFG parser. We deliberated using some other parsers as well, but stuck with the Stanford parser for its accuracy on sample input. We also use some processing modules from the NLTK parser [7] which is in Python and provides a natural interface to our system, but for the primary parse we use the Stanford parser. Some other parsers we considered were the Link parser [8] and MontyLingua [9]. The Link parser provides dependency parsing, however with some sample runs it did not seem as accurate as the Stanford parser which is trained on a large corpora. The Link parser also provides multiple parses of the

sentence, however the Stanford parser just provides the most likely parse of the input making our parsing task simpler. MontyLingua was developed at the MIT Media Lab and used in the Metafor project, however MontyLingua is a pretty basic parser with very little support for grammatical extraction. The parser is also not very accurate on complicated sentences and does not generate a tree parse.

Since the Stanford parser is in the Java language, we created a separate server for the parser. The server takes HTTP requests with the input sentence and extracts all the necessary parses (stemming, tagging, tree and dependencies). It uses Wordnet [10] for stemming the input sentence. We use stemmed words to allow unique indexing for objects, both for the purpose when the user references those words with different word forms (e.g. cried, crying or dog, dogs) and for retrieving commonsense knowledge about those objects and verbs from the database.

The Stanford parser generates parse trees which are imported into the NLTK Tree class for further processing. We wrote a chunk extractor for extracting over parse trees. This gave us the best of both worlds—the world of a good chunk extractor as well as that of an accurate parser. The program generator works only on the final form of the input sentences rather than on partial chunks. The benefit of using the final forms of sentences is that new forms of input can be easily added by adding its grammatical form to the grammar library. A typical extractor definition looks like this:

```
OBJ_PROP=
(S
  (NP-np1 )
  (VP (VB? is)
    (ADVP-normally* (RB 'normally|usually))
    (NP|NNS-np2*)(ADJP-state2*)(PP-state2*)))
```

This particular form parses state descriptions and object properties. The "-np1" token signifies extraction. The "?" token signifies a wild card. The "*" token signifies optional chunks or repeatability, the quote " ' " signifies compulsory matches and the "|" signifies either/or between the words or syntactic classes. The grammar descriptions were built by analyzing typical

English sentences in the Stanford parser. Other parsers might have slightly different outputs, so there would have to be a bit of redefinition of the grammars to allow the chunk extractor extract from the parses generated by another parser. One can certainly optimize a lot of areas in the parser and grammars, but this particular form worked well for the purpose of integrating it with other areas of the system.

The frame parsing parses using regular expressions. A typical definition is

```
FP_PRINT=(?:say|print|announce) "([a-z\d\s\.\,]+)"
```

which parses the output statements. Often, chunks extracted from the frame parses are run through the grammar parses to further identify programming artifacts. All the grammar and frame parsing definitions are provided in the appendix. Let us now look at specific cases of the input and the how we extract information from it to generate code and do other representational changes.

2.3.1 Object creation, properties, states and relationships

"There is a microwave oven on the table. It is empty."

This is a typical statement that can be used to create objects in a MOO room. In this case a microwave oven and a table is created. An `on(table,microwave oven)` relationship is added between the microwave oven and the table. Furthermore, the oven's state is identified as empty. In some inputs, users reference uninstantiated objects. In these cases objects are added automatically.

"It is big.", "The switch is on.", "It is empty."

These statements modify an object that was created earlier. Note that "big" is a property and "on" and "empty" are states. At this time we do not make a distinction between properties and states, however, generally properties are more permanent than states even though both can affect object class associations.

For the statement, “It is empty.”, the parser resolves the reference to the oven which is the subject of the statement. In case the parser identifies an ambiguous reference it can generate a clarification dialog to the user who can then select between the possible referenced objects. For the case of the statement “The switch is on”, the system identifies that the switch is in an 'on' state (or property).

For the statement "the switch is on", let us say that the system knows from previous input that switches are normally off and on and off are mutually exclusive properties for switches. In such a case, the system makes the initial state of the switch "on" and removes the normal "off" state of the switch. The system keeps track of such states and only allows one of them to be active at a time.

Certain states are commonly addressed using a past participle—for example "a broken cup" or "the button is pressed". Such cases are important because users could say statements like "When you drop a delicate cup, it breaks" and "You cannot put milk in a container, if the container is broken". In this case, the MOOIDE identifies that the "break" verb will make the state of the cup to be the past participle of 'break' i.e. "broken" and this state will be added to the state list of the cup. Now, not all verb past participles are not used as state identifiers e.g. "thrown cup" doesn't really make much sense. The way we handle this issue is that past-participle identifiers are used as second class adjectives and are promoted to first class adjectives (printed out in object descriptions) only if there is some rule or piece of code that refers to the past participle. Since it is unlikely that somebody would refer a cup with the phrase "thrown cup", we expect that the system would avoid description errors for such adjectives.

2.3.2 Verb definitions

"You can bounce the ball."

"You can put food in the basket."

"There is a basket on the floor in which you can put clothes"

Verb definitions are parsed to create verbs that MOO users can invoke to interact with objects. In the above case, the "bounce" verb is attached to the object ball and the user can activate it by entering "bounce ball" which will activate the code in the verb.

In the second case, the "put" verb is attached to the "basket" object. However, in this case the verb accepts a variable argument and the user can enter "put bread in basket". The system automatically generates code to check if the "bread" object is of the "food" type, otherwise it generates an error message saying "You can only put food in the basket". MOOs also have standard object-oriented hierarchies that are available in programming languages. However, every programmer would need to build his own system of hierarchies. In MOOIDE object hierarchies for common objects are automatically available (one would need to populate using a common standard word hierarchy like that of ConceptNet/Cyc or Wordnet). Hierarchies are important because they are a big part of human commonsense procedural communication.

The above patterns only define verb command formats and some basic argument rules. More complicated rules are also parsed and they are discussed in the next section. To generate program code we parse patterns of the form "When you press the button,..." in which everything after the verb argument structure is sent to the code parser. This is discussed in more detailed in the section after next.

2.3.3 Verb argument rules

Verb argument rules specify rules that put restrictions on the execution of a verb. These rules are typically of the form "You can/cannot [only] VP, [only] condition". The condition is optional. Let us look at the specific cases of these rules and how they are handled.

"You can put water in the water bucket."

This statement adds the standard code for 'put' to the water bucket—i.e. after the 'put water in water bucket' command is executed, `in(water bucket,water)` would be True.

"You can only put bread in the toaster."

This rule makes sure that only objects of the class "bread" are allowed to be put into the toaster object. If somebody tries to put other kinds of objects in the toaster, the system would automatically generate a message saying "You cannot do that because you can only put bread in the toaster". Notice the rule is preceded with the phrase "You cannot do that because".

"You cannot put liquids in a toaster."

"You cannot put things inside a locked container."

These statements works similar to the previous case and negate object classes. The error message is similar - "You cannot do that because you cannot put liquids in a toaster." Technically one *can* put liquids in a toaster. One can extend the above semantics to allow statements like "You *should not* put liquids in a toaster" which would generate a warning message but still allow the action.

In MOOs, often there are objects with warnings like "*the toaster dares you to put water in it*" suggesting an action that could have dire consequences in normal life. In Inform 6 there is a similar message - "*That dangerous act will achieve little*" which can be used as a polite refusal to forbid execution of prohibited actions. One can perhaps use negative rules from a commonsense database to suggest new kinds of verbs to users.

Notice that the error messages seem very commonsensical because they would normally be acquired through a human being either through acquisition by a system like OpenMind or through MOOIDE's own commonsense acquisition process—unlike descriptions of logical decision—the logic compiled by a reification process which would not always be easy to understand.

"You can put food in an oven, only if the oven is empty."

"You can only press the button, if it is not stuck."

These statements allow arbitrary conditionals to be attached to the rule—the LHS of the rule will only be valid if the RHS is True. MOOIDE parses a variety of conditionals; they are described later.

2.3.4 Verb program generation

Arguably the most important part of creating verbs is being able to describe arbitrary code for execution. An example:

When a person presses the start button, if there is food in the oven,

the oven makes the food hot and gives it to the person.

If there is no food in the oven, say "nothing to heat, sire".

The description of verb code starts with statements of the form "When/If [action description], [code description]". Note that the statement “If a person presses...” instead of “When a person presses...” is also parsed. The dialog manager maintains a program pointer to ensure that the code at placed in the right in the code tree. The pointer can be manually or automatically set. Users can also enter statements of the form:

To empty a container, take its contents and put them on the floor.

The pattern "To [verb description], [code description]" is normally a response to a dialog question which was in response to an unknown verb command like "When someone presses the

button, empty the container." The system does not know how to execute an "empty" verb on the "container" object. So the dialog manager responds with:

> *"I do not know how to empty a container. You can describe it to me by saying - To empty a container... ".*

To which a user can respond by describing a new verb with a statement of the form above. If one notices, these kinds of verb descriptions are very much like function definitions.

Alternatively, in the above case, the user might be requesting a command "empty oven" on an 'oven' object, and the system has a way to empty a container. The system also knows that "ovens are containers". In this case, MOOIDE will automatically use the "empty" verb from the container object and use it on the 'oven' object.

2.3.5 Decision constructs

Humans do not normally explicitly communicate deeply nested decision trees. Inferring decision nesting from their descriptions is therefore a hard problem. In human communication decision structures are derived from dialog in which the describer expects the receiver to either know or ask about what to do under default conditions and should inconsistencies be detected by either party, there would be further discourse to resolve them. Although we would have liked to, we have not implemented a formal dialog model to help users elaborate complex decision trees. In normal human conversations decision delimitation and organization is automatically inferred from the semantics of the statements leaving out the need to use statements like "ENDIF". Which is why, we do not think it would have been useful to build an interface just for the sake of allowing movement of code in a decision tree—the decision parameters of which would have to be quite explicitly defined by users. It would have been simple to implement such an interface, but it goes against the philosophy of this project. The system should use commonsense to structure code in a decision tree.

The dialog for decision constructs is manged using the following heuristic assumptions about decision statements:

1. After the definition statement for the IF construct, every statement that is required to be nested between the IF and the Else should be preceded by a "Then" or "Also" or "And" or "And then". The first statement that does not have either of these prefixes will cause the program pointer to be reseted to the parent level of the previous IF/Else construct.

2. Every statement between an IF and Otherwise is automatically nested. This is a form of error correction should the user forget to put the prefixes.

MOOIDE's decision statements take a straightforward form. An IF construct is parsed by the frame "IF [condition], [code]". After the IF statement the program pointer is pointed right after the [code] statement. Let us illustrate this with an example. Let us say the user enters these sequence of inputs:

```
When someone presses the dispensing button, if
the amount of money in the vending machine
counter is equal to 75 cents and there is a candy
in the vending machine, give the candy to the
person.
Then say "Enjoy the candy".
Otherwise if there is no candy in the vending
machine, say "We've run out of wares". Then give
back the money to the person.
Otherwise, say "You need to put exactly 75 cents
in the vending machine so I will give the money
back to you".
Then give back the money to the person.
If there is no money in the vending machine, say
"Please put money in first".
```

This code is structured like this:

```
When (someone presses the button) :
      IF (amount is 75 cents and there is a candy) :
            Give(person,candy)
            Say "enjoy the candy"
      ElseIF (not in(vending machine,candy):
            Say "We've run out of wares"
            Give(person,money)
      Else
            Say "You need to put exact 75 cents.."
            Give (person,money)
      IF (there is no money in the vending machine):
            Say "Please put money in first.
```

The logic of the statements above is self explanatory. However, note the statements starting with "Then...". Both statements start with 'Then' to avoid resetting the program pointer to a level above the "IF" construct. Note that ideally, the first and the second IF construct should be exclusively executed to each other because they deal with the True and False case of the condition `in(vending machine,money)`? However, human descriptions can often leave out such details expecting them to be inferred from the semantics of the input. We have not attempted to solve this problem.

2.3.6 Decision conditions

MOOIDE parses a variety of decision conditions. Although one can describe arbitrary sets of conditions using the boolean variables "and/or", we only parse one pair of such conditions. Let us look at some examples of the kinds of conditions we parse:

Boolean

Existential	If [there is food in the oven]
Property	If [the oven is empty]
Object class	If [it is a food] / [is a type of food]

Comparatives can have "is/are less than", "is/are equal to" and "is/are greater than" between the compared statements.

Count	If [the number of drinks in the fridge] is less than 5
	If [there are 5 drinks in the fridge]
Variables	If [the money] is equal to
Functions	If [the color of the box] is

For parsing, first the frame parser isolates "and/or" connectives. If the conditionals are boolean properties, the parser writes code for their evaluation. If the conditionals are comparatives like "is equal to" it parses the statements on both sides of the conditional and generates code accordingly. Note in statements of the form "When you put food in the oven, If it is hot", the anaphora "it" is dynamically resolved to verb argument "food" because it does not directly resolve to a known object.

2.3.7 Verb commands

Users can create verb commands. These are typically of the form:

"Press the button", "Give it to the person"

"Make it hot and give it to the person"

Verb commands can be interpreted as function calls. MOOIDE comes with a built in set of commands - put object (in/on/over/into) object, give object to object, take object from object and remove object. The put/give/take commands change the `in(parent,child)` and `HasA(parent,child)` relationships. The remove command removes an object from the system. Since users can refer to the `'in'` relationship using `'hasa'` (e.g. if there is food in the oven, one can say if the oven has food), if a user asks about a `'hasa'` relationship, the system also checks for the `'in'` relationship.

Although we have not implemented exception handling, just like function calls in programming languages, verb commands can also raise exceptions. Note that return values would need to be semantically analyzed rather than explicitly programmed. For example, a person might just say:

```
When you press the start button, if the oven is
unplugged say "sorry I cannot cook cause I don't have
energy."
Otherwise make the contents of the oven hot.
To cook food, put food in the oven and press the start
button.
```

In the above case, there is no way for the "cook" verb to know weather it will succeed or fail. One can notice that failure conditions are implicit in the semantics of verb definition. Users can also say statements of the form:

```
When you try to press the start button, if the
oven is unplugged say "sorry I cannot cook cause
I don't have energy".
```

In this case, the "try" word before the verb phrase "press the start button" identifies to the failure condition—it would be normal to assume that code definition is when the "start button" cannot be "pressed" because of some rule. In such a case, an exception can be raised so that it can be catched by another set of statements. In the above case one can then write:

```
To cook food, put food in the oven and press the
start button.
If you are unable to cook, say "sorry cannot
cook".
```

In ideal cases, the system should check for verbs that generate exceptions by parsing or simulating commonsense rules. Often, failure messages would need to be disabled in verb calls in the case of exceptions because it would seem as if the system is leaving a debug trace.

2.3.8 Iterators, variables and loops

Take the contents of the box and print their name. Then put them in the bag.

Make all the objects in the oven hot.

The "Take" command simply returns the objects that are referred to and puts them in a temporary variable that can be referenced in future statements. Our first iteration was to write a grammar for common loops in the form of "For every object in the box, ...", however these do not seem to be in common use in English language. We have tried to identify statements that are more naturally used in daily communication.

2.3.9 Messages

Messages are typical MOO messages. MOO is predominantly a text based environment so messaging would be the most popular statement in the MOO. Unlike other verb commands, messages have a special syntax:

Say/Print/Announce "message"

Say "the bin has" followed by the number of items in the bin." followed by "items in it".

Sometimes users might want to print messages with arguments. In this case one can put the 'followed by' phrase to put arbitrary arguments in the message statements.

2.3.10 Anaphora resolution

In MOOIDE, anaphora references are normally allowed in every input. When a noun phrase is encountered, it is passed to the reference resolver which returns a reference to an instance of an object in the system – which can be one of:

- An instantiated object in the MOO world definition e.g. *"the cat is blue"*

- A commonsense generic object e.g. *"cats are blue"*

- Program argument e.g.:

"When you put food in the oven, if the food is hot ... "

- Locally instantiated variable e.g.:

"take the contents of the oven, and put them in the bag"

Often, resolution can be done using a bit of commonsense (although we do not do this in MOOIDE)—as in this case:

"When you put food in the oven, if it is empty, say ... "

In this case, foods do not normally have the 'empty' property, so "it" should be resolved to the oven and not the food argument object. Similarly in the case of affordances:

"When you try to press it... "

In this case, we know that buttons can be pressed and not ovens. So this verb definition resolves to the button object rather than the oven object.

We have used centering [21] rules as guidelines to do reference resolution.

1. For each utterance in a discourse there is precisely one entity that is the center of attention.

2. There is a preference, formalized as Rule 2, (1) for consecutive utterances within a discourse segment to keep the same entity as the center of attention, and (2) for the entity most prominently realized in an utterance to be identified as the center of attention.

3. The center of attention is the entity that is most likely to be pronominalized: this preference is formalized as Rule 1.

The context manager maintains a list of most recently referenced objects and the context in which they were referenced. When a reference arises (both anaphoric and object references), previous references in the same context are looked up and resolved accordingly. Let us look at a few examples to illustrate the algorithms:

```
There is a button.
It is on the oven.
You can press it.
```

In this case, all the three "it" references are resolved to the 'button' object which is the subject of the first sentence. Furthermore, references of the same form are resolved to the same object like in this example:

```
There is an oven.
There is a button on it.
There is a keypad on it.
```

In this case, "it" (oven) is in the object role of the sentence, but it is kept at the center of attention in the last two inputs. However, if a user added this sentence to the above input:

```
When you throw something in a fire, it gets burnt.
```

In this case, "it" is resolved to 'something' which is the 'verb argument' which will be dynamically resolved when the code is executed. For example if the user says "throw paper in fire", "it" will be resolve to the "paper" object.

2.4 Commonsense features

MOOIDE incorporates a variety of commonsense features many of which have been explained in the previous section because they integrate with the working of the parser and intermediate with the code generator. Let us summarize them here:

Object hierarchies: Objects are arranged in object hierarchies. Objects can have properties that can be mutually exclusive. Objects can have normality properties as well. Object properties are inherited as long as they are not overridden by child objects.

Verb affordances: Verbs are defined over objects. There are rules that can be defined over the ability of the verb to be executed over a certain object. These rules are also inherited into child objects. Verb arguments in rules obey object hierarchies.

Cause effect rules: Cause effect rules are standard pieces of code like "When an object is eaten, it disappears."

Verb suggestions: Verbs on parent objects and on object instances are suggested to users at the creation of an object.

Acquisition: Generic facts that users provide into the interface are collected and can be vetted and reused later.

2.4.1 Learning commonsense knowledge

Certain sentence frames of the form "cats are animals" indicate universal commonsense knowledge. These will be acquired into the database so that they can be assumed attached to new objects that users create. One needs to however, distinguish between facts that are be specified in a virtual world and those that are available in a database like ConceptNet. Facts in ConceptNet are expected to be universally true, unlike facts in a virtual world—which can be bent for entertainment purposes. An example is someone might create a "talk" verb on a "cat" object in a virtual world. From this example one should not generalize that "all cats can talk". In fact the correct fact is that "no cat can talk". Therefore all facts are collected and entered into a temporary

database to be periodically presented to the user through our interface to be vetted before they are added to the database.

Facts are acquired by parsing user input. Such facts are of two types—direct facts and derived facts. An example of a direct fact is "all cats are animals". An example for a derived fact is "One can feed a cat"—derived from a verb creation input of the form "When a person feeds the cat,...". Direct facts are assumed to be universally true and are entered into the database automatically. They are however, still presented to the user at the time of fact vetting, who can select any fact that was miscollected and remove it from the database. Derived facts are not assumed to be universally true and are not automatically entered. At the time of vetting, the user is presented with these facts and he has to select facts that are true, only after which they are added into the database. Since sentences entered into MOOIDE are added to the commonsense database only after they are successfully parsed, MOOIDE's commonsense database collects facts and reuses them in English instead of a separate representation like a semantic network or logical predicates.

2.4.2 Suggesting verbs

While users are elaborating their virtual worlds we use dialogs to suggest verbs to users. Facts about object verbs stored in the commonsense database are extracted to generate a dialog of the form: "I see you created an NP". "My knowledge base says that you can do different things like—VP the NP". Where the first NP is the object created and "VP the NP" are all the verbs that were extracted from the commonsense database. This particular dialog gives ideas to the user to create code for any of those verbs. The user can also request the system to show verbs for an earlier object by issuing the command "What can you do to NP ?" which will trigger the system to retrieve verbs from from the commonsense database and show them to the user similar to as described above.

2.4.3 Universal verbs and properties and cause-effect rules

Universal commonsense verbs and properties are automatically attached to objects without asking users. Anecdotally, users of Inform have expressed frustration with the need to enter trivial commonsense facts into its stories. We solve this issue by externalizing the commonsense database. With integration with a database like ConceptNet, if an oven only allows objects of the class food to be cooked, a user would not need to specify that rice is a type of food, so that the verb allows it to be put into the oven. Facts of the form "When delicate objects are dropped, they break." are cause effect facts and are also acquired into the database.

2.5 Dialogs and messages

In the previous examples we have mentioned places where messages are generated and different types of dialogs are used. We have not built a formal dialog model in the spirit of projects like Collagen. Programming in natural language would require a complex model of conversation between the user and the programming interface. Without the dialog model, the system would be mostly an elaborate parser and would lose out on really probing the user to elaborate the environment he had in his mind. We do have some features though that we share with features of dialog models—reference resolution, output messages, context management etc. The list of messages that are generated is given in the appendix. Let us summarize the dialogs that are generated.

Reference resolution: This dialog is generated when the system identifies an unresolved reference error. The user is notified of that error and is given with a list of previous objects and he can select from those objects which one he really referenced in the input.

Verb definition: This dialog is generated when a user references a verb command that has not been defined. If the user does not define the verb, the system adds the past-participle of the verb to the state of the object that the verb acts upon. It also generates a message stating that the verb was executed.

Verb suggestions: MOOIDE maintains a list of verbs associated with objects in the commonsense database. When a new object is added, it gives the list of those verbs to the user.

Chapter 3

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 in a reasonable amount of time. Since such subjects are not expected to be experienced programmers, a natural language interface provides them an alternative to learning a programming language.

Our goal, therefore is to evaluate whether they can use our interface without getting frustrated, possibly enjoying the interaction while successfully completing a test programming scenario.

3.1 Experiment hypothesis

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 (once or twice during the whole test scenario).

3.1 Experiment protocol

1. Fill out the consent form.
2. Get introduction to games and MUDs
3. Perform familiarization scenario in MOOIDE.
4. Perform task scenario.
5. Fill out post-test questionnaire.

3.2 Familiarization scenario

We first run 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 complete sequence of the familiarization scenario is provided in the appendix.

3.3 Experiment task scenario

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):

```
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 out interesting messages when the lever
is turned. So if you're turning the lever, the machine
might say "oooh I malfunctions"
It also says out 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.

3.4 Post test questionnaire

	Question	Type
1	It is enjoyable interacting with MOOIDE.	Likert
2	MOOIDE a good way to add new characters and behaviors into to the MUD.	Likert
3	If I was playing this game with a few people, I would use MOOIDE to introduce new characters and behaviors into the MUD.	Likert
4	I feel it might be easier to use MOOIDE that to learn programming in a formal language.	Likert
5	I would show MOOIDE to my friends or kid brother/sister as a way to introduce him/her to programming.	Likert
6	What would you want to improve in MOOIDE?	Text
7	Any other suggestions or overall comments.	Text

3.5 Study results

We conducted the study with 10 subjects. All the subjects had minimal or no formal programming experience. The age range for the subjects varied between late teens to mid-forties. All except two participants were native English speakers. Two female subjects who were of Chinese origin were fairly fluent in English. The subjects were 70% female vs. 30% male. All subjects were able to able to complete the scenario within 5 min which was significantly less than our original expected time of 20 min. Summary of the questionnaire:

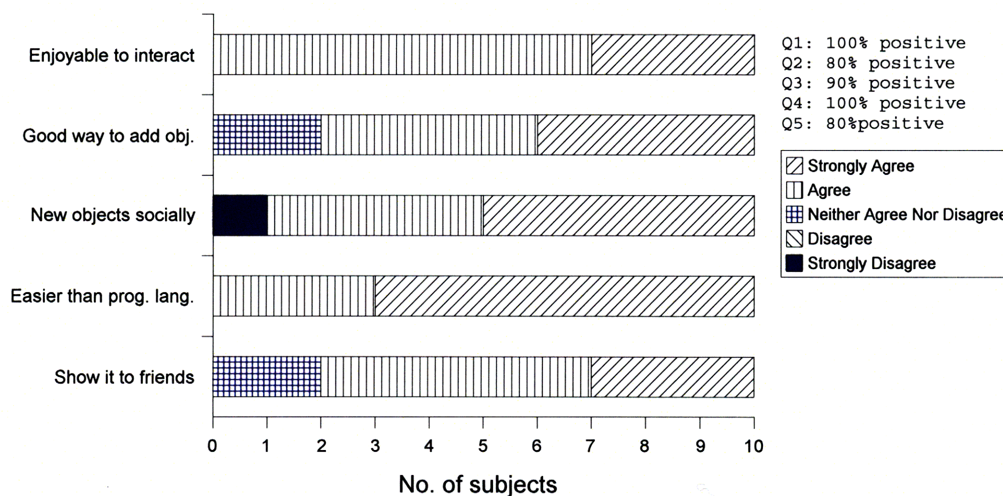


Figure 9 : Post Evaluation Questionnaire Summary

3.6 Observations

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 (which is a 3rd party MOO simulation that we integrated into MOOIDE but did not program ourselves) 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 would 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 3rd party simulation interface.) 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 an 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".

During the evaluation, sometimes when a subject would stop to think before typing in an input, it was not clear whether he was thinking about behavior or about articulating that behavior or about syntactic issues in MOOIDE. One will need separate research to understand the first two of these issues through wizard-of-oz kinds of studies.

Chapter 4

Discussion

During the course of the design, development and evaluation of MOOIDE, we came across many ideas and issues that cut across different segments of study. These areas include MOO and Interactive fiction environments, commonsense reasoning, user interface design, natural language parsing etc. Each of these areas are well developed areas of research. As we try to put together a system that uses ideas from each of them, we generate many other ideas which often bring to our attention larger issues and questions that need more thought and investigation. In this section we will try to summarize some of the larger issues that we ran into during the course of this research.

4.1 Commonsense in MOO environments

Adding commonsense capabilities to MOOs and other interactive fiction environments generates a number of interesting questions. MOO environments are often fantasy environments in which its users tend to be more daring than the real world. We discussed about an earlier example in which the statement "*you should not put liquids in toasters*" is an example of a commonsense fact that could potentially expose interesting behavior making the interaction much more exciting. Perhaps we need to distinguish between commonsense facts that users expect the system to know and those commonsense facts that would generate interesting behaviors if overridden in the MOO fantasy world. One can also imagine people exploring situations to see the effect of violating commonsense to see what happens. Such kind of simulations are not possible in the real world because of the dire consequences that can result when one "*tries to put liquid in a toaster*".

Does commonsense and natural language interaction allow MOO systems to have semantics closer to the real world? For example no common MOO/Interactive fiction

environment allows people to refer to events back in time. During our preliminary evaluation, we came across cases where people were entering inputs like "*when you put a coin in the machine AND kick the machine,...* ". This kind of a case is an event chaining example and although we did not model event chaining in a formal way, we extended the parser to allow these kinds of inputs and generate simple behavior from it. It would be hard to expect that a commonsense powered MOO system would have comprehensive models of all kind of commonsense microtheories [27]. It would also be impossible to expect for the MOO system to build new microtheories from user collected facts. That means, in some ways, that a commonsense powered MOO programming system would have certain limitations, but the limitations need to be communicated to its users and programmers. How would the interface communicate the limitations on MOO semantics to its users?—it would be rather user-un-friendly to generate an error message of the form "*I do not understand event chaining...*".

Commonsense itself has its issues when one uses natural language to express it. There are a large number of commonsense facts that people do not articulate at all. Perhaps they do not even have a simple way articulating them in natural language. For example when one "*throws a cookie on a person it falls down and it is not 'on' the person after that*". However, "*if one throws something on a table, it would normally remain 'on' the table*". Cognitive linguists and computer scientists have worked on the problem of verb semantics and modeled various aspects of that using frame based representation and logical rules. Cyc, VerbNet, FrameNet etc. have used different approaches to solving the problem of expressing such commonsense facts. However, when one wants to express and acquire these facts using natural language, it becomes hard to do so because people do not normally communicate these facts to each other. It almost seems unnatural and superfluous to express them (although we did see one of the subjects express a statement "*if you put a coin in the candy machine, there will be a coin in the candy machine*"). So in such a natural language programming environment, the question arises—what kind of expectation should the interface give to the user? How does the user know how much

commonsense does the system have? Should the user be expressive of every little detail (as in the case of common programming languages)? Perhaps there is a middle ground. We do not know.

4.2 Learning programming

We do not feel that people in general who do not have programming experience are great at describing complicated procedures. Learning programming is not just about learning syntax, but really it is getting one to become a better articulator of procedures. Just asking people to "describe a vending machine" would not be the best way to get them to be elaborative enough so that the system could generate a sufficiently detailed program. People do not program objects in daily life and it would be presumptuous to expect that they appreciate the difference between "just describing something in English" vs "programming the verb behaviors on an object". We found that running our subjects through a simple scenario in which one shows how programming needs some amount of detailing and it is a slightly different activity than just describing the world gets them to understand how one would go about programming in the first place.

Another question a person would have would be "Why would I want to do 'programming'. Not only is it a geeky activity, I don't know what I can do with it". When one runs him through an actual example, he sees the creative results and feels internally rewarded. MOO environments, as opposed to other programmable environments are also social environments where different people interact with each other. The social dimension is important for the segment of population that is not inclined to creative activities purely for self pleasure. Building something and showing it to friends gives social status which is a important motivator and cannot be ignored.

4.3 Understanding natural language

As a natural language interface, one cannot help but wonder about the current limitations that computers have in understanding natural language. Certainly, computer program are not at a stage where they can maintain a steady conversation about common human activities. However, progress is being made in the different issues in building interfaces with natural language

abilities. Humans are however adaptable and if there is a broad coverage of linguistic syntax in a natural language interface, we feel they will be able to use a system like ours without getting frustrated. For example we have tried to cover different syntaxes of verb commands like "the person has a candy" to "the machine gives him the candy". With further research, one can expand upon the pattern library to improve the natural language capabilities of MOOIDE. There are other areas that would need further research for example dialog modeling for programming and building a library of common procedures that would integrate with MOOIDE. Furthermore as we mentioned earlier, programming a system like MOO does not require complicated spatial reasoning or an understanding of a huge variety of verb argument expressions. These constrains also significantly reduce the size of the natural language understanding problem.

So, how much natural language understanding is enough for the purpose of programming a MOO like environment? This is a hard question to answer and is related both with the expectation that the system sets upon the user as well as the proficiency of the user to express procedures. We would expect that people who are regular MOO users to be better able to appreciate that in MOOs, programming tasks are limited to building objects and giving them verb commands. Over-specification of non-behavioral information is not required. We would also expect that they would therefore be comfortable with a less sophisticated natural language understanding/dialog system. But then would their expressivity be reduced if they program in a natural language interface that would normally hide some of the programmability of the MOO as opposed to when they program in a programming language that would normally expose a very large bit of the programmability of the MOO? Again, perhaps there is a middle ground where the natural language dialog system can calibrate on the fly how much complexity of the programming task it should it expose to its user.

4.4 Natural language programming for formal languages

One can also ask the question—how about giving a natural language interface to common programming languages like C++ or Java which program environments like client software on a desktop machine or large web services on the Internet. Let us look at some broad issues that arise when one thinks about the kinds of skills human programmers have when it comes to programming complicated procedures. An ideal interface for natural language programming would be similar to a human programmer and would need to have the following three core skills that complement each other.

Programming expertise: This involves the ability to write and manipulate code for algorithms, design patterns, common data structures etc. These standard skills are those that are acquired in a typical college computer science program. In addition to skills acquired during formal training, an expert programmer learns a large number of other techniques during his years of experience in solving programming problems in a variety of contexts. These techniques involve the manipulation and adaptation of standard procedures and data structures for use in multiple analogous contexts.

For example, a programmer might have learned to write a sorting algorithm to sort numbers—say *QuicksortNum*. To adapt QuicksortNum to work on strings, he only needs to adapt the comparator function in the sorting algorithm enabling him to use *QuicksortNum* to alphabetically sort a list of names. Such abstraction may not need to be taught by a teacher—human intuitive capabilities provide sufficient intelligence to make the programmer do these kinds of adaptations himself.

The constraints of programming language syntax like data typing and functional argumentation also provide capabilities through which programmers iteratively adapt programs to different problems. To illustrate the above point, let us say a programmer at the first intuitive leap might put string arguments into the comparator function of *QuicksortNum*. The compiler would then generate an error saying that the comparator function only allows numbers and cannot

compare strings. This error would cause the programmer to go through reflective error resolution which would help him to generate ideas to solve the above problem—bringing out another procedure that he learnt—the string comparator procedure—which he will then add to the original *QuicksortNum* procedure to make a *QuicksortStrings* procedures. This new procedure can now be canned and used in the future.

It might have been possible that the programmer did not have an abstracted comparator function for numbers but rather had a block of code that had the purpose of generating a decision (less than, equal or greater than) about comparing numbers. In such a case, the programmer will have to change individual IF constructs such that they use his string comparator function. He will also have to ensure that the IF decisions correctly map to the return values of his string comparator function (for example the string comparator function might return -1 for less than, 0 for equal and +1 for greater than. In this case the "IF num1<num2" boolean statement will map to "IF strcmp(str1,str2)=-1"). In the above example, we see that both high level intuitive capabilities with lots of big and small ideas and low level code manipulation skills are required for a well functioning programming assistance.

As illustrated above, standard programs and procedures that programmers learn in their initial years are modified and augmented through experience to build a larger library of procedures that can be used in a variety of new situations.

Elaborating and communicating design requirements: The second skill involves helping users to communicate and elaborate design requirements and help them make design decisions. This is a task we humans excel at because through our commonsense reasoning skills we can manage partial, underspecified and conflicting information. We are able to use dialogs to come to joint decisions about the specification. Today a large number of programming environments exist—from assembly language programming on computer chips to operating system shell scripting to high level assistants like the Mac Automator. MOOs are also programmable with many languages. The interface would need domain specific knowledge about the environment and

should be able to speak in the vocabulary of the environment. It might even have to explain cryptic technical jargon in simple English. It would need domain agnostic communication skills to communicate what it understood from the user and any standard solutions it has that can approximate the user's problem. The user might like the standard solution, but with certain modifications—which would requires code manipulation skills. Elaborating the program specification in these environment becomes a joint task between a human and the programming interface.

Domain specific and commonsense knowledge: The third skill is the ability to understand and manipulate domain specific expertise and domain-general commonsense knowledge. Domain general commonsense knowledge consists of common worldly facts like"ovens are electrical appliances". And rules like "you can put objects in containers, only if they are empty." Domain specific knowledge is always built upon commonsense knowledge and so a large commonsense database is a requirement. Domain specific knowledge is expected to be of a smaller size and there might be a variety of sources through which it can be acquired. An interesting exercise would be to take a common programming environment—say MATLAB and build a parser for its documentation and help files to an intermediate representation that would acquire MATLAB vocabulary and the syntax of its programming constructs and idioms. This would be one way in which domain specific knowledge could be loaded into a natural language programming assistant.

For MOO programming however, we do not feel that one needs to provide an interface that exposes the expressivity of common programming languages. A sufficiently developed interface that handles a good amount of syntactic varieties can for the time being get people good familiarity with programming and get them to appreciate the thrill and excitement that programmers feel as creative agents.

Chapter 5

Related Work

The closest work that this work builds upon is the Metafor system [2] which aimed to be a visualization tool for code rather than a natural language programming system. It takes simple natural language program descriptions from users and uses simple parsing to generate scaffolding code fragments. Our system also started with identifying code fragments in parses, but went much further to identify and elaborate many more programming constructs in natural language. Metafor did not directly generate code into any environment. MOOIDE integrates a programmable environment and generates Python code that runs in that environment. Metafor explored the idea of parsing programming constructs in English to help people build broad outline of the code from their articulations, but it did not aim to provide a full natural language interface to programming. Metafor used commonsense from the ConceptNet database for disambiguation tasks in natural language. MOOIDE integrates commonsense reasoning in other ways into its interface—for example it allows users to describe affordance rules to objects and it is able to resolve object classes in verb arguments during run-time. These features are again available through MOOIDE's natural language interface which acquires commonsense facts into its database during the interaction. MOOIDE also has a significantly expanded sets of commonsense frames that it parses.

The Inform series of programming languages in the area of Interactive Fiction has tried give people an easier language in which to describe MOO-like worlds. Inform7, the latest version has a grammar that allows its programmers to use English-like descriptions. Inform is still a fairly strict programming language albeit it feels and reads a lot like English. Architecturally, MOOIDE is significantly different because it does not have any formal grammar. MOOIDE works by pattern matching on the inputs. So, parser improvements happen by adding more patterns to the

parser. The main benefit of not having a formal grammar is that MOOIDE can do partial matching and use dialogs to get users to elaborate on the parts it did not understand. The other upgrade that we have given to the natural language programming features of Inform7 is the ability to use standard acquirable commonsense knowledge in the programming effort.

The Scratch programming language [11] is a highly visual language targeted to children. It tries to make programming fun and interactive. The visual interface of Scratch allows people to represent program structures as hierarchical blocks. MOOIDE generates Python code, however it would be quite interesting to generate code in the Scratch format because visual feedback in Scratch is more intuitive to understand. However, the kinds of programs that people make in Scratch are also visual in nature and visual commonsense can be quite complicated to parse and understand.

Pane and Myers studied non-programmer's solutions to programming problems [12]. Non-programmer subjects were given representatives of common programming tasks. They were presented with programming tasks and asked to solve them on paper using whatever diagrams and text they wanted to use. PacMan video game was chosen as a source of interesting problems for this purpose.

Quoting from the study: "The majority of the statements written by the participants were in a production-rule or event-based style, beginning with words like if or when. However, the raters observed a significant number of statements using other styles, such as constraints, other declarative statements (that were not constraints) and imperative statements. The participants' solutions seem to be more reactive, without attention to the global flow of control. When imperative statements were used, it was usually for local flow of control. The declarative style seems to have been primarily used for setting up the scenario (data, characters, objects, etc.) of the program." [12]

The study resulted in the creation of a new programming language called HANDS [13]. HANDS featured an event based programming style that closely matched the problem solutions

in the study. HANDS, however is still a programming language with a conventional syntax and does not allow natural language input. MOOIDE takes ideas from the study and also parses a variety of declarative statements for object descriptions and event style descriptions for behaviors. It also allows people to use rules to describe various constraints that could be put on the behaviors of objects. These inputs are however not confined to a grammatical syntax like that of a conventional programming language.

Mihalcea, Liu and Lieberman have previously [14] analyzed some aspects of natural language processing for programming – focusing on steps and loops and developed techniques to map linguistic constructs onto program structures. A variety of syntactic correspondences between natural language and programming structures have been identified. For example the SVO construction is mapped to a method-argument or agent-method structure. They have also described ways to find steps and loops in program descriptions. They evaluated the algorithms by creating a gold standard of 25 programming descriptions by manually labeling the main steps and repetitive structures in the descriptions. MOOIDE parses some of these structures, but does not go into significant details about loops mainly because MOOs do not typically have complicated repetitive behaviors.

There were many previous systems that were built in the broad area of automatic programming/natural language programming. The KBEmacs system [15] which was part of the programmer's apprentice project was capable of using a series of high-level commands to generate or modify a program. It did not use natural language the way MOOIDE does. The U-Tel [16] system tried to address the user-task elicitation problem so that domain experts could describe task descriptions which the system could parse into steps and sub-task hierarchies. The U-Tel user interface also allowed experts to annotate text with sequencing details. The task elicitation problem is similar to the problem of describing detailed procedures in MOOIDE. U-Tel, however did not generate program code and was only a tool to help articulate the correct series of steps in a procedure.

The Collagen framework [17] built upon the SharedPlan theory [18] provides a framework for intelligent user interfaces in which the user interface is aware of and keeps track of its user's goals and intentions. Collagen has been applied to multiple projects in the area of consumer electronics. Although we did not build a formal dialog model, ideas in Collagen can be particularly useful once they are applied to managing programming discourse. Now that MOOIDE has articulated a broad array of actions that are required to be modeled in a natural language programming system, one can take that action model and build a discourse model around it similar to that available in Collagen. Collagen builds upon extensive empirical and modeling work in discourse processing and human collaboration. The SharedPlan theory of Grosz and Kraus provides a formal model for modeling mutual beliefs about goals and actions of collaborating participants as well as their capabilities, intentions and commitments.

Appendix

Appendix – I : Grammars and Parse Frame Definitions

Object Creation and Relationships

```
(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP-np1 )))

(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP-np1 )))

(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP
      (NP-np1 )
      (PP (IN-rel in)
        (NP-np2 )))))

(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP
      (NP-np1 )
      (SBAR
        (WHNP (WDT which))
        (S
          (NP (PRP you))
          (VP (MD can)
            (VP (VB?-verb press))))))))

(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP
      (NP-np1 )
      (SBAR
        (WHPP (TO|ON|IN-vprep in)
          (WHNP (WP|WDT which)))
        (S
          (NP (PRP you))
          (VP (MD can)
            (VP (VB?-verb cook)
              (NP-varg ))))))))

(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP
      (NP-np1 )
      (PP (IN-rel on)
        (NP
          (NP-np2 )
          (SBAR
            (WHNP (WDT which))
            (S
              (NP (PRP you))
              (VP (MD can)
                                (VP   (VB?-verb
press))))))))))

(S
  (NP (EX 'there))
  (VP (VB? 'is)
    (NP
      (NP-np1 )
      (PP (IN-rel in)
        (NP
          (NP-np2 )
          (SBAR
            (WHPP (IN|TO|ON-vprep on)
              (WHNP (WDT|WP which)))
            (S
              (NP (PRP you))
              (VP (MD can)
                (VP (VB?-verb write)
                  (NP*-varg ))))))))))

(S
  (NP-np2 )
  (VP (VB? 'has)
    (NP
      (NP-np1 )
      (PP* (IN-rel in)
        (NP )))))

(S
  (NP-np2 )
  (VP (VB? 'has)
      (NP-np1 )))

(S
  (NP
    (NP-np1)
    (ADVP (RB normally)))
  (VP (VB? have)
    (NP
      (NP-np2)
      (PP* (IN|TO in)
        (NP)))))
```

Object States

```
(S
  (NP-np1 )
  (VP (MD can)
    (VP (VB? 'be)
      (ADVP|ADJP (RP|JJ-state1 on)
        (CC or)
        (RP|JJ-state2 off)))))

(S
  (NP-np1 )
  (VP (VB? 'is|are)
     (ADVP-normally* (RB normally))
    (NP-np2*)(ADJP-state2*)(PP-state2*)))
```

```
(S
  (NP-np1 )
  (VP (VB? is)
    (ADVP-normally* (RB normally))
    (PP (IN|TO|ON-rel in)
      (NP-np2 ))))
```

Verb affordances

```
(S
  (NP (PRP 'you))
  (VP (MD 'can) (RB-cannot* not)
    (ADVP* (RB-only only))
    (VP (VB?-verb put)
      (NP-varg )
      (PP (IN|TO|ON-vprep in)
        (NP-np )))))


(S
  (NP (PRP 'you))
  (VP (MD 'can) (RB-cannot* not)
    (ADVP* (RB-only only))
    (VP (VB?-verb put)
     (NP
      (NP-varg )
      (PP (IN|TO|ON-vprep in)
        (NP-np ))))))

(S
  (NP (PRP you))
  (VP (MD can) (RB-cannot* not)
    (ADVP* (RB-only only))
    (VP (VB?-verb press)
      (NP-np ))))


(S
  (NP-np )
  (VP (MD* can) (RB-cannot* not)
    (ADVP* (RB-only only))
    (VP (VB?-verb cook)
      (PRT* (RP out))
      (NP-varg* )
      (PP* (TO to)
        (NP (PRP you)))
      (ADVP* (RB-only only))
          (SBAR (RB-only* only) (IN|WHADVP-
type )
        (S-chunk
          )))))

(S
  (NP-np )
  (VP (MD* can) (RB-cannot* not)
    (ADVP* (RB-only only))
    (VB?-verb cook)
      (PRT* )
      (NP-varg* )
      (PP* (TO to)
        (NP (PRP you)))
      (ADVP* (RB-only only))
          (SBAR (RB-only* only) (IN|WHADVP-
type )
        (S-chunk
          ))))

(S
  (NP-np)
  (VP (MD 'can) (RB-cannot* not)
    (VP (VB be)
      (VP (VB?-verb petted)))))

(S
  (NP-np)
  (VP (MD* 'can) (RB-cannot* not)
    (VP (VB?-verb give)
    (NP-varg) (PP*))))
```

Verb definitions

```
(SBAR|SBARQ
  (WHADVP|IN )
  (S|SQ
    (NP )
    (VP (VB?-verb puts)
      (NP-varg )
      (PP (IN|TO|ON-vprep into)
        (NP-np )))))


(SBAR|SBARQ
  (WHADVP|IN )
  (S|SQ
    (NP )
    (VP (VB?-verb puts)
     (NP
      (NP-varg )
      (PP (IN|TO|ON-vprep into)
        (NP-np ))))))

(SBAR|SBARQ
  (WHADVP|IN )
  (S|SQ
    (NP )
    (VP (VB?-verb presses)
      (NP-np ))))


(SBAR|SBARQ
  (WHADVP|IN )
  (S|SQ
    (NP-varg)
    (VP (VB? is)
      (VP (VB?-verb put)
        (PP (IN|TO-vprep into)
          (NP-np))))))


(SBAR|SBARQ
  (WHADVP|IN )
  (S|SQ
    (NP-np)
    (VP (VB? is)
      (VP (VB?-verb pressed)))))
```

Conditionals for IF constructs

```
(FRAG|X
  (NP-np ))

(NP
  (NP )
  (PP (IN of)
    (NP-np )))
```

```
(NP
  (NP )
  (PP (IN of)
    (NP
      (NP-np1 )
      (PP (IN|ON-vprep in)
        (NP-np2 )))))

(S
  (NP (EX there))
  (VP (VB? is)
    (NP
      (NP-np2)
      (PP (IN|ON-vprep in)
        (NP-np1)))))

(S
  (NP-np2)
  (VP (VB? 'is)
    (PP (IN|ON-vprep in)
      (NP-np1))))


(S
  (NP-np1)
  (VP (VBZ 'has)
    (NP-np2
      (NP-np2)
      (PP (IN-vprep in)
        (NP (PRP it))))))

(S
  (NP-np1)
  (VP (VBZ 'has)
    (NP-np2)))

(S
  (NP-np)
  (VP (VB? is) (RB-cannot* not)
    (ADJP|PP (JJ|RP-state empty))))


(S
  (NP
    (NP-np1 )
    (PP (IN|ON on)
      (NP-np2 )))
  (VP (VB? is) (RB-cannot* not)
    (ADJP (RP|RP-prop off))))
```

Verb commands

```
(S
  (VP (VB?-verb give)
    (ADVP* (RB back))
    (NP-np2 )
    (ADVP* (RB back))
    (PP* (TO|IN-vprep to)
      (NP-np1 ))))

(S
  (VP (VB?-verb 'make)
    (S
      (NP-np1)
      (ADJP (JJ-prop hot)))))

(S
  (NP-np1)
  (VP (VB?-verb 'becomes)
    (ADJP (JJ-prop hot))))
```

```
(S
  (NP )
  (VP (VB?-verb 'makes)
    (S
      (NP-np1)
      (ADJP (JJ-prop hot)))))

(S
  (NP )
  (VP (VB?-verb gives)
    (NP-np2)
    (PP (TO|IN-vprep to)
      (NP-np1))))

(S
  (NP )
  (VP (VB?-verb gives)
    (NP-np2)
    (NP (ADVP* (RB back))
    (PP (TO|IN-vprep to)
      (NP-np1)))))

(S
  (NP )
  (VP (VB?-verb gives)
    (NP-np1)
    (NP-np2)))

(S
  (NP )
  (VP (MD 'will)
    (VP (VB?-verb give)
      (NP-np1)
      (NP-np2))))


(S
  (NP-np2)
  (VP  (MD 'will) (VP (VB?-verb heats)
    (NP-np1))))

(S
  (NP-np2)
  (VP  (MD 'will) (VP (VB?-verb gives)
    (NP-np2) (PP (TO)(NP-np1 (PRP you))))))

(S
  (NP-np2)
  (VP (VB?-verb heats)
    (NP-np1)))

(S
  (NP (DT the) (NN oven))
  (VP (VBZ heats)
    (NP
      (NP (DT the) (NN food))
      (PP (IN inside)
        (NP (PRP it))))))

(S
  (VP (VB?-verb Put)
  (ADVP* (RB back))
    (NP
      (NP-np2)
      (ADVP* (RB back))
      (PP (IN|TO-vprep in)
        (NP-np1)))))
```

```
(S
  (VP (VB?-verb give)
    (ADVP* (RB back))
    (NP-np2)
    (ADVP* (RB back))
    (PP (TO|IN-vprep to)
      (NP-np1))))

(SINV
  (VP (VB?-verb gives))
  (ADVP* (RB back))
  (NP-np2)
  (ADVP* (RB back))
  (PP (TO|IN-vprep to)
    (NP-np1)))

(S
  (VP (VB?-verb remove)
    (NP-np1)))

(VP (VB?-verb make)
  (NP-np1 )
  (PRT|ADJP-state ))

(S
  (VP (VB?-verb make)
    (S
      (NP-np1)
      (PRT|ADJP-state))))

(S
  (NP-np1 )
  (VP (VB?-verb turns)
    (PRT|ADJP (RP|JJ off))))
```

Miscellaneous

```
(S
  (NP-np1 )
  (VP (VBP are)
    (NP-np2 )))

(S
  (NP-np1 )
  (VP (VB?|VBP is)
    (NP
      (NP (DT* a) (NN|NNS type))
      (PP (IN 'of)
        (NP-np2 )))))

(S
  (NP-np1 (JJ|VBG*) (NNS cats))
  (VP (VBP are)
    (ADVP (RB usually))
    (ADJP|PP (JJ|IN-prop small))))
```

NP Parser

```
(NP (CD*) (PRP-ref*) (DT-det*) (VB?|JJ|NN?-
name*))
```

Frame parse definitions in regular expressions

Commands

```
FP_CMD="([!a-z\d\s\.\,\'\"]+) (?:and then) ([!a-z\d\s\.\,\'\"]+)"
FP_CMD2="([!a-z\d\s\.\,\'\"]+) (?:and|then) ([!a-z\d\s\.\,\'\"]+)"
```

Output Messages

```
FP_PRINT="(?:say|print|announce) "([!a-z\d\s\.\,\']+)""
FP_PRINT1="(?:[a-z\s\d]+)(?:says) "([!a-z\d\s\.\,\']+)""
FP_PRINT2="(?:say|print|announce) "([!a-z\d\s\.\,\']+)" (?:and) ([!a-z\d\s\.\,\'\"]+)"
FP_PRINT3="(?:[a-z\s\d]+)(?:says)     "([!a-z\d\s\.\,\']+)"     (?:and     then)     ([!a-
z\d\s\.\,\'\"]+)"
```

When/If

```
FP_WHEN="(when|if) ([a-z\s\d]+), ([!a-z\s\d\"\,\.\']+)"
```

Affordances

```
FP_YOUCANCOND="(you can|you cannot) ([a-z\s\d]+),(?: |)(only |)([a-z\s\d\"\,\.]+)"
FP_YOUCAN="(you can|you cannot) ([a-z\s\d]+)"
FP_YOUCANCOND1="([a-z\s\d]+),(?: only )([a-z\s\d\"\,\.]+)"
```

Commands

```
FP_CMDRULE="([a-z\s\d]+) ([a-z\s\d]+),(?: |)(only |)([a-z\s\d\"\,\.]+)"
```

Conditions and IF

```
FP_IF_STMT="(?:and |)if ([a-z\s\d]+), ([!a-z\s\d\"\,\.\']+)"
FP_IF="(?:and |)if ([a-z\s\d]+)"
```

```
FP_ANDOR="([a-z\s\d\"]+) (and|or) ([a-z\s\d\"]+)"
FP_EQ="([a-z\s\d\"]+) (equals|is equal to|is smaller than|is less than|is greater than)
([a-z\s\d\"]+)"
```

Miscellaneous

```
FP_ISA="([a-z\s\d\"]+) is a ([a-z\s\d\"]+)"
FP_IS="([a-z\s\d\"]+) is ([a-z\s\d\"]+)"
FP_NUMOF="the number of ([a-z\s\d\"]+)"
FP_FIRST="first ([a-z\s\d\"\,\.]+)"
FP_THEN="then ([a-z\s\d\"\,\.]+)"
FP_OTW="otherwise ([a-z\s\d\"\,\.]+)"
FP_FOR="for ([a-z\s\d\"\,\.]+)"
FP_UNTIL="until ([a-z\s\d\"\,]+)"
```

Appendix – II : Evaluation Sequence

Introduction to games and MUD games

3-D First person shooter games vs. role playing games

Multi user dungeons and dragons

Game programming

Create text based objects

Give them text based behaviors (verbs)

Example: Text based objects

Creating a cat

```
@create $thing as cat
```

Creating a tickle verb

```
@cmd cat.tickle <obj> calls cat_tickle()
@newfunc cat.cat_tickle(self)
.
```

MOO> tickle cat

Creating a pet verb

```
@cmd cat.pet <obj> calls cat_pet()
@newfunc cat.cat_pet(self)
      print "purrrrrrrrrrrrr"
.
```

What does MOOIDE do?

There is a cat

You can tickle the cat

When you tickle the cat, it says "Purrrrrrrrrrrrr"

>tickle cat

Training Scenarios

Type in these statements:

There is an oven.

The oven has a start button

There is chicken in the kitchen

You can press the start button

You can put chicken in the oven

Click on "TEST"

"put chicken in oven"
"press start button"

Type in this statement:

When you put chicken in the oven, say "you can press the start button to cook the food"

Click on "TEST"

"put chicken in oven"

Type in these statements:

When you press the start button, if there is chicken in the oven, make it hot and give it to the person.

Otherwise say "you need to have chicken in me first"

Click on "TEST"

"press start button"

"put chicken in oven"

"press start button"

Type in these statements:

You can eat the chicken, only if it is hot.

Click on "TEST"

"eat chicken"

"press start button"

"eat chicken"

Short Test/Training Case

Create a dog that yelps when you kick it

Create a toaster with a button and that toasts bread and gives it to you.

when you put bread in the toaster and press the button, the toaster toasts the bread and gives it to you

When you put water in it, it says "oh man I will not be able to work anymore"

Final Test Case: Candy machine Scenario Description

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 out interesting messages when the lever is turned. So if you're turning the lever, the machine might say "oh you need to feed me with a magic coin first"

It also says out interesting things when magic coins are put in it like "ooooh I malfunctioned"

And finally when you kick the machine, it gives the candy.

Bibliography

1. Schiano, D. J. (1999). *Lessons from LambdaMOO: A social, text-based virtual environment.8(2), 127-139.*
2. Liu, H., & Lieberman, H. (2005). Metafor: Visualizing stories as code. *IUI '05: Proceedings of the 10th International Conference on Intelligent User Interfaces,* San Diego, California, USA. 305-307.
3. Nelson,G. (2006). Natural language, semantics and interactive fiction. Available at: http://www.inform-fiction.org/I7Downloads/Documents/WhitePaper.pdf
4. Singh, P., Lin, T., Mueller, E. T., Lim, G., Perkins, T., & Zhu, W. L. (2002). Open mind common sense: Knowledge acquisition from the general public. *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002,* 1223-1237.
5. Liu, H., & Singh, P. (2004). ConceptNet --- A practical commonsense reasoning tool-kit.*22*(4), 211-226.
6. Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics,* Sapporo, Japan. 423-430.
7. Loper, E., & Bird, S. (2002). NLTK: The natural language toolkit. *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics,* Philadelphia, Pennsylvania. 63-70.
8. Sleator, C. D., & Temperley, D. (1993). Parsing english with a link grammar. *In Third International Workshop on Parsing Technologies,*
9. Liu, H. (2004). MontyLingua v2.1 Free Natural Language Understanding Toolkit and API available at: http://web.media.mit.edu/~hugo/montylingua/
10. Miller, G. A. (1995). WordNet: A lexical database for english.*38*(11), 39-41.
11. Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004). Scratch: A sneak preview. *C5 '04: Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing,* 104-109.
12. 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.
13. Pane, J. F. (2002). A programming system for children that is designed for usability. Carnegie Mellon University.
14. Mihalcea, R., Liu, H., & Lieberman, H. (2006). NLP (natural language processing) for NLP (natural language programming). *CICLing,* 319-330.
15. Rich, C., & Waters, R. C. (1988). The programmer's apprentice: A research overview.*21*(11), 10-25.
16. Tam, R. C., Maulsby, D., & Puerta, A. R. (1998). U-TEL: A tool for eliciting user task models from domain experts. *IUI '98: Proceedings of the 3rd International Conference on Intelligent User Interfaces,* San Francisco, California, United States. 77-80.
17. Rich, C., Sidner, C. L., & Lesh, N. (2001). Collagen: Applying collaborative discourse theory to human-computer interaction.*22*(4), 15-25.

18. Grosz, B. J., & Sidner, C. L. (1986). Attention, intentions, and the structure of discourse.*12*(3), 175-204.

19. Liu, H., & Lieberman, H. (2005). Programmatic semantics for natural language interfaces. *CHI '05: CHI '05 Extended Abstracts on Human Factors in Computing Systems,* Portland, OR, USA. 1597-1600.

20. Liu, H., & Lieberman, H. (2004). Toward a programmatic semantics of natural language. *Proceedings of VL/HCC'04: The 20th IEEE Symposium on Visual Languages and Human-Centric Computing. September 26-29, 2004,* 27-30.

21. Grosz, B., Weinstein, S., Joshi, A. (1995). Centering: a framework for modeling the local coherence of discourse, *Computational Linguistics,* v.21 n.2, p.203-225

22. Lebling, D., Blank, M., Anderson, T. (1979) Zork: A Computerized Fantasy Simulation Game, *IEEE Computer,* 12:4, 1979: 51-59.

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

24. Falsetti, J., & Schweitzer, E. (1995). SchMOOze University: A MOO for ESL/EFL students. *M. Warschauer (Ed.), Virtual connections: On-line activities and projects for networking language learners*, (pp. 231-232). Honolulu: University of Hawai'i, Second Language Teaching and Curriculum Center.

25. Lenat, D., Guha, R. (1989) Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project, 1st. Addison-Wesley Longman Publishing Co., Inc.

26. Minsky, M. (2000). Commonsense-based interfaces. *Commun. ACM,* 43, 8 (Aug. 2000), 66-73.

27. Lenat, D., Guha, R. (1991). The evolution of CycL, the Cyc representation language, *SIGART Bull.,* 2, 3 (Jun. 1991), 84-87