Metafor: Visualizing Stories as Code

Hugo Liu
MIT Media Laboratory
20 Ames St., Cambridge, MA, USA
hugo@media.mit.edu

Henry Lieberman
MIT Media Laboratory
20 Ames St., Cambridge, MA, USA
lieber@media.mit.edu

ABSTRACT

Every program tells a story. Programming, then, is the art of constructing a story about the objects in the program and what they do in various situations. Traditionally, these stories are expressed in so-called *programming languages*. These languages are easy for the computer to accurately convert into executable code, but are, unfortunately, difficult for people to write and understand.

In this paper, we explore the idea of using descriptions in a natural language like English as a representation for programs. While we cannot yet convert arbitrary English descriptions to fully specified code, this paper shows how we can use a reasonably expressive subset of English as a *visualization tool*. Simple descriptions of program objects and their behavior are converted to *scaffolding* (underspecified) code fragments, that can be used as feedback for the designer, and which can later be elaborated. Roughly speaking, noun phrases can be interpreted as program objects; verbs can be functions, adjectives can be properties. A surprising amount of information about program structure can be inferred by our parser from relations implicit in the linguistic structure. We refer to this phenomenon as *programmatic semantics*. We present a program editor, Metafor, that dynamically converts a user's stories into program code, and in a user study, participants found it useful as a brainstorming tool.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: *interaction styles, natural language;*

General Terms

Design, Human Factors, Languages, Theory.

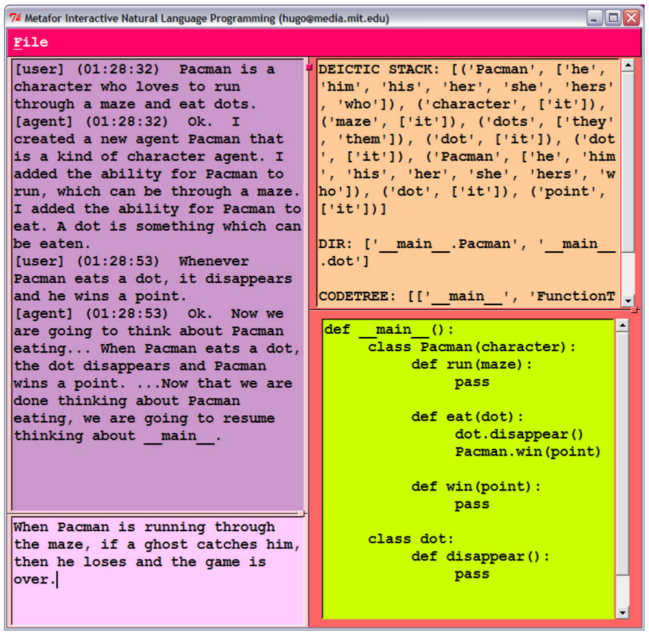
Keywords

Natural language programming, case tools, storytelling

1. INTRODUCTION

Computer programming is usually a harrowing experience for the uninitiated. It is difficult enough to achieve minimum proficiency in it, but to truly master programming – to attain an intuitive and almost philosophical understanding of its flow, and to reach the point of being able to easily articulate arbitrary *thinkable* ideas within that framework – well, few ever reach this point. Yet there is a sense that those who have absorbed programming into personal intuition have gained new tools for thinking, discovering the ability to articulate any *procedural* idea with algorithmic rigor.

We have developed an intelligent user interface which we hope will inspire changes to the way that computer programming is learned and practiced. Metafor is an interface for visualizing a person's interactively typed stories as code. As a person types a story into Metafor, the system continuously understands the narrative, interpreting it programmatically using a theory of the programmatic semantics of natural language, and updating a side-by-side "visualization" of the person's narrative as *scaffolding code.* The visualized scaffolding code may not be directly executable, but rather, it is meant to help a person reify her thoughts in terms of programmatic elements. We believe that Metafor is a novel system which can accomplish at least two main *goals*: 1) The goal of assisting novice programmers in developing intuitions about programming; and 2) The goal of facilitating intermediate programmers with system planning by serving as a brainstorming and "outlining" tool (just as writers outline ahead of writing).


Metafor Interactive Natural Language Programming (hugo@media.mit.edu)
File

```
[user] (01:28:32)  Pacman is a
character who loves to run
through a maze and eat dots.
[agent] (01:28:32)  Ok.  I
created a new agent Pacman that
is a kind of character agent. I
added the ability for Pacman to
run, which can be through a maze.
I added the ability for Pacman to
eat. A dot is something which can
be eaten.
[user] (01:28:53)  Whenever
Pacman eats a dot, it disappears
and he wins a point.
[agent] (01:28:53)  Ok.  Now we
are going to think about Pacman
eating... When Pacman eats a dot,
the dot disappears and Pacman
wins a point. ...Now that we are
done thinking about Pacman
eating, we are going to resume
thinking about __main__.
```

```
When Pacman is running through
the maze, if a ghost catches him,
then he loses and the game is
over.
```

```
DEICTIC STACK: [('Pacman', ['he',
'him', 'his', 'her', 'she', 'hers'
, 'who']), ('character', ['it']),
('maze', ['it']), ('dots', ['they'
, 'them']), ('dot', ['it']), ('dot
', ['it']), ('Pacman', ['he', 'him
', 'his', 'her', 'she', 'hers', 'w
ho']), ('dot', ['it']), ('point',
['it'])]

DIR: ['__main__.Pacman', '__main__
.dot']

CODETREE: [['__main__', 'FunctionT
```

```
def __main__():
    class Pacman(character):
        def run(maze):
            pass

        def eat(dot):
            dot.disappear()
            Pacman.win(point)

        def win(point):
            pass

    class dot:
        def disappear():
            pass
```

Figure 1. Metafor's user interface. Clockwise from the lower left corner, the four windows display the narrative being entered, the dialog history of the person-to-system interaction; an *under-the-hood* dump of Metafor's current memory (for demo and debugging purposes only: not shown to beginning users); and the code visualization of the story, currently being rendered as Python code (although rendering engines can exist for any language).

1.1 Cultivating Intuition and Facilitating Brainstorming

This tool may help novice programmers to more rapidly develop intuitions about programming because the immediate feedback provided by the system allows the novice to focus on the *mappings* between their ideas naturally expressed as story, and the code which is a direct consequence of those ideas. Rather than "book-learning" programming concepts, Metafor affords novices the opportunity to learn experientially, as is often advocated by progressive education researchers.

For intermediate programmers, Metafor provides a way to create an early brainstorm or outline of a project at a very high level of description. Just as writers are accustomed to creating brainstorms and outlines before they set out on their first draft, programmers may also benefit from this phenomenon. The goal of outlining is to help an author better focus on fleshing out ideas about the task without being distracted prematurely by the imposition of the rules of form which something must obey. In programming, as in writing, it is advantageous to flesh out the details of the task *before* actual programming begins, because once a person is bogged in the concerns, and demands of programmatic syntax, and bugs, and the commitments of representational choice, it can be very difficult to switch back and forth to thinking about the task, or to undo representational choices already made.

1.2 Context of this Work

The larger context for this work is our overarching goal of enabling programming by natural language. Previously we performed some feasibility studies for programming by natural language (Lieberman & Liu, 2004a) by examining how fifth graders naturally expressed the task of programming Pacman via storytelling. We have also been exploring how natural language might inherently be interpretable under a programmatic semantics framework (Liu & Lieberman, 2004b). The current system represents our progress toward the larger goal, but we believe that Metafor's goal of producing *scaffolding code* as immediate feedback to a story is compelling in and of itself for its potential applications to education and improving programming praxis.

1.3 Scope and Limitations

We should emphasize that Metafor *cannot* convert completely arbitrary English into fully specified code. Our parser cannot understand every grammatically legal English construction. And, although our parser does use a large knowledge base of common sense knowledge, discussed below, it doesn't know everything a programmer might think of saying. It's not difficult to "fool" it, and our goal is not to get 100% coverage. However, we do believe that the scope of its understanding will be sufficiently large as to be usable in practice. We are encouraged by experience with MUDs and other text-based interaction games, that achieve usable interaction even with very simple template-based parsers; with natural language interfaces to databases and search engines; and with conversational "chatbots". As we will see, we are bringing far more sophisticated analysis to the table than these systems typically use.

We, of course, will try to set user's expectations with admonitions to "keep it simple". We are assuming that the user has at least a passable reading knowledge of the programming language. Since the goal is for the user to watch how each statement affects the generated program, it is easy to spot mistakes in translation and undo them. We also provide an introspection facility, discussed in Section 4.1, that allows a user to see, for a given piece of code, what natural language expressions can be generated. This helps the user to get a feel for what the translator is capable of. We are actively exploring other ways of making the translation process fail-soft. We think that computer science has been so reluctant to use modern natural-language technology in interfaces, for fear of making mistakes, that the field has ignored important opportunities to make interfaces significantly easier for people, especially beginners, to use.

1.4 Paper's Organization

The rest of this paper is structured as follows: First, we dive into an extended interaction with Metafor to give the reader a better sense for the system's capabilities. Second, we expound on a theory of programmatic semantics for natural language that is at the core of Metafor's interpretive abilities. Third, we briefly survey the implemented system. Fourth, we share the results of a user study we performed with non-programmers and intermediate programmers on the subject of brainstorming. Fifth we present a discussion of related works. We conclude by recapitulating the contributions of this work.

2. AN EXTENDED INTERACTION

This section presents an extended interaction with Metafor on an example taken from the world of MUDs (multi-user dungeons) and MOOs (multi-user object-oriented), which are text-based virtual realities popular on the Internet. We chose to illustrate Metafor in this domain in particular because MOOs *are* themselves interactive stories, where the characters and even inanimate objects, are programmable. A typical MOO consists of text descriptions of "rooms". When a (human) player enters a room, she sees a description such as,

```
You are in a tiny room with a desk in the center of the
room. On the desk there is a pen and a stuffed teddy
bear. In the corner is a mouse hole. A mouse sticks its
head out.
```

Users can say text or "emote", leading to dialogs like the following:

```
        Miranda gives you a hug
        Mouse says, "I'm here to hug you!"
        Mouse hugs Miranda
        Mouse says, "I made a mistake"
```

Characters in MOOs can be programmed with simple scripts, expressed in a "English-like", though formal, programming language, the following from (Bruckman, 1997).

```
Stacy is a frendly killer whale. She has Brown eyes and
her tail has a rash.
1 script on Stacy:
on flap this number "times"
        set flapped to number
        if flapped > 5 times
                emote " blinks her eyes happily"
        endif
end
```

MOOs boast many avid non-programmer and beginning programmer gamers amongst their ranks, and we imagine that something like Metafor might one day (soon) be used to enable them to augment and program the virtual realities themselves, just by telling the story of how things should work.

For reasons of space, we cannot include the system agent's dialog responses, nor can we display the contents of the *under-the-hood* debug window. The visualized code seen here is rendered in the syntax of the Python programming language, but in principle, renders can be written for any other language. A few peculiarities of Python benefit from explanation: "def" is the function declaration symbol, and "pass" is a placeholder keyword for a function or class with an empty body. Code which is changed or added since the previous sentence is highlighted in red.

(1) There is a bar with a bartender who makes drinks.

```
class bar:
    the_bartender = bartender()
class bartender:
    def make(drink): pass
```

This one sentence actually unfolds into four declarations: "There is bar," "bar has a part called bartender," "there is a bartender" (implied), and "bartender can make drinks." Note that "who" is resolved as "the bartender" and not the bar because "bartender" is animate (some limited semantic knowledge is used for this).

(2) The bar has a menu containing some drinks, which include: a sour apple martini, a margarita, and rum and coke.

```
class bar:
    the_bartender = bartender()
    the_menu = menu()
class bartender:
    def make(drink): pass
class menu:
    drinks = [sour_apple_martini, margarita, rum_and_coke]
class drink: pass
class apple_martini(drink):
    properties = ["sour"]
class margarita(drink): pass
class rum_and_coke(drink): pass
```

"Drinks" is recognized as a list because it is determined by "some," and its declared contents are in list structure. Each of its member elements automatically inherit the base class "drink." "Sour" was plucked from the object name "sour apple martini" and set instead as a property because it is the interpreter's default policy that object names are noun phrases, and adjectival modifiers are properties.

(3) The sour apple martinis and margaritas are sweet, while rum and coke is bitter.

```
class bar:
    the_bartender = bartender()
    the_menu = menu()
class bartender:
    def make(drink): pass
class menu:
    drinks = [sour_apple_martini, margarita, rum_and_coke]
class drink: pass
class apple_martini(drink):
    properties = ["sour","sweet"]
class margarita(drink):
    properties = ["sweet"]
class rum_and_coke(drink):
    properties = ["bitter"]
```

Note here how the property "sweet" distributes over "sour apple martinis" and "margaritas," instead of forming a single object called "apple martini and margarita;" this is because the interpreter pre-recognizes existing objects because parsing.

(4) When a customer orders a drink, the bartender tries to make it. When the bartender is asked to make a drink, he makes it and gives it to the customer only if the drink is in the menu's drinks; otherwise, the bartender says "sorry i don't know how to make that drink" to the customer.

```
class bar:
    the_bartender = bartender()
    the_menu = menu()
class bartender:
    def make(drink):
        if (drink in menu.drinks):
            bartender.make(drink)
            bartender.give(drink, customer)
        else:
            bartender.say( \
"sorry i don't know how to make that drink", customer)
    def give(drink, to_customer): pass
    def say(quote, to_customer): pass
class menu:
    drinks = [sour_apple_martini, margarita, rum_and_coke]
class drink: pass
class apple_martini(drink):
    properties = ["sour","sweet"]
class margarita(drink):
    properties = ["sweet"]
class rum_and_coke(drink):
    properties = ["bitter"]
class customer:
    def order(drink):
        bartender.make(drink)
```

Expressions begun with "when" usually specify the scope of the following utterance. Possessives like "menu's drinks" act just like dot notation for scope in object-oriented programming. Pronouns like "it" (inanimate, singular) and "he" (animate, male, singular) are resolved in the system's preprocessing of discourse deixis. Although there might be other representational choices for "say," "make," and "give," the interpreter's default behavior is to treat the main verb as the predicate. Note that with additional semantic knowledge such as commonsense, there are more choices.

(5) When the customer asks the bartender to choose, the bartender makes a random sweet drink from the menu if the customer's age is under 30; or else the bartender makes a sidecar.

```
class bar:
    the_bartender = bartender()
    the_menu = menu()
class bartender:
    def make(drink):
        if (drink in menu.drinks):
            bartender.make(drink)
            bartender.give(drink, customer)
        else:
            bartender.say( \
"sorry i don't know how to make that drink", customer)
    def give(drink, to_customer): pass
    def say(quote, to_customer): pass
class menu:
    drinks = [sour_apple_martini, margarita, rum_and_coke]
class drink: pass
class apple_martini(drink):
    properties = ["sour","sweet"]
class margarita(drink):
    properties = ["sweet"]
class rum_and_coke(drink):
    properties = ["bitter"]
class customer:
    age = None
    def order(drink):
        bartender.make(drink)
    def ask_bartender_to_choose():
        if customer.age < 30:
            bartender.make(random.choice(
                filter(lambda drink:
                  'sweet' in drink.properties,
                  menu.drinks)))
        else:
            bartender.make(sidecar)
```

Sentence (5) demonstrates the difference between the declarative and procedural contexts. While in the declarative modes of sentences (1), (2), or (3), an utterance of the form "ADJ NN" would have created "NN" as an object annotated with the "ADJ" property, there is a different handling under a procedural context (cued by a "within-function" scope and a reference to an existing object i.e. "drink"). The procedural interpretation of "ADJ NN" is "select the NN objects with the property JJ." "Random" is an implemented primitive in the Metafor interpreter, and in Python.

Hopefully what this interaction demonstrated is that 1) natural language is particularly economical in the amount of information it contains; that 2) natural language is elegant in reusing the same constructions e.g. "ADJ NN" under different intentional contexts (e.g. procedural versus declarative) to accomplish different goals (i.e. declaration, versus relational selection); and finally 3) the ambiguity of natural language's representational choices (e.g. "make(drink)" or "make_drink()") is actually quite a virtue of flexibility which most popular programming languages today do not yet enjoy.

We also hope that this interaction begins to illustrate the systematicity and some of the regularities of the Metafor interpreter. More will be said on the interpreter in the presentation of the system implementation in Section 4. However, before arriving there, Section 3 describes a theory of the programmatic semantics for natural language which is driving the interpreter; this theory will synthesize together much of what was seen in the sample interaction.

3. A PROGRAMMATIC SEMANTICS FOR NATURAL LANGUAGE

Natural language, be it English, Chinese, or Russian, shares basic structure, and basic protocols for communication (but in this paper, we consider only English). Although the subdivision of Artificial Intelligence which tries to computationalize the understanding of natural language, called Narrative Comprehension or Story Understanding, usually represents stories using thematic role frames, Schankian scripts, Jackendoff trajectory space, or otherwise (a great review of the field given in (Mueller, 1999)), there is fundamentally no reason why natural language cannot be interpreted as if it were a programming language.

3.1 Basic Features

In fact, there are many reasons to believe that natural language already *implies* a natural programmatic interpretation. The way in which natural language tends to reify concepts as objects with properties or personify concepts as having capability begins to resemble a style of agent-programming. The natural role of nouns and noun phrases as objects (e.g. "the martini"), adjectives as properties (e.g. "*sweet* drinks"), non-copular verbs corresponding to functions (e.g. "*make* a drink"), and verb arguments as function arguments (e.g. "give *the drink* to *the customer*") resembles the organization of object-oriented programming. Natural language also has a system of inheritance (e.g. "a martini is a *drink* …"), as well as conventions for reference which closely resemble dot notation (e.g. "The customer's age" ←→ customer.age). A more protracted discussion about these basic programmatic features of natural language can be found in (Liu & Lieberman, 2004b); however, the rest of this section will be dedicated to a more advanced discussion of natural language's programmatic semantics, including the dispelling of some falsities about language, and a review of some of the elegant programmatic features of natural language which go beyond basic features of most programming languages. We cluster these discussions around a few hot topics.

3.2 "Ambiguity"

Mapping from natural language into programming language introduces "ambiguity" (enclosed here in scare quotes because the word is often used in a derogatory manner) which formal programming languages are often not accustomed to. While some see the inherent ambiguity of natural language as a problem, we see it as an important advantage.

Conventional programming is hard in no small part because programming languages force a programmer to make inessential decisions about representation details far too early in the design and programming process. When those early decisions later prove ill-advised, the messy and error-prone process of *refactoring* and other program modification techniques become necessary. By using natural language understanding to construct the mapping between natural language specifications and concrete programming language details on a dynamic basis, we retain representational flexibility for as long as it is needed. For example, consider the utterance, "sour apple martini;" there is some ambiguity in how this object should be represented and what should be parameterized. For representational simplicity, we might first reify it as "`class sour_apple_martini`." However, upon later encountering a "`sweet apple martini`" and a "`sour grape martini`" and applying some background world knowledge that "sweet" and "sour" are flavors and "grape" and "apple" are kinds of fruit, we might revise the representation of "sour apple martini" to be better parameterized:

```
class martini:
    def __init__(self,flavor='sour',fruit='apple'):
        self.flavor, self.fruit = flavor, fruit
```

An affordance of relating programs as stories is that we can continually reinterpret the story text as evidence crops up for better representational choices.

3.3 Representational Equivalence

The fact that the representation of an object can change so fluidly, yet stay consistent with the goals of the task is quite remarkable, and the sort of representational dynamism found in natural language is quite unparalleled by any formal programming language. Consider the following statements and the revision of representation which ensues.

```
a) There is a bar. (atom)
b) The bar contains two customers. (unimorphic list)
c) It also contains a waiter. (unimorphic wrt. persons)
d) It also contains some stools. (polymorphic list)
e) The bar opens and closes. (class / agent)
f) The bar is a kind of store. (inheritance class)
g) Some bars close at 6pm. (subclass or instantiatable)
```

In formal programming languages, representational revisions b) through g) are potentially quite costly, because in natural language, the revisions are quite natural. In creating a flexible representation of natural language which steers short of the representational commitments demanded by rigid programming languages, we create a representationally neutral structure for an object which is used in Metafor. It is a tuple of the form:

```
(full_name, arguments, body)
```

A type inspector examines the contents of the arguments and body, and dynamically assigns the object a type. If the constitution of the arguments or body changes, so will the type. If for example, an object's body contains only two similarly typed elements, then it is a list. If the body also contains functions, it is a class, and so forth. The effect of this on interpreting story as code is that as new utterances reveal new details, the representation of the code will be able to adapt.

A slightly more tricky equivalence phenomenon is *nominalization* (turning any adjective into a noun). For example "The drink is sweet" and "The drink has sweetness" are equivalent, although in the latter, the property is talked about as an object. The way to interpret this is to assume simplicity, and only add complexity where necessary. So at first “sweetness” is just a property of “drink,” but if “sweetness” is elaborated as an object or agent (e.g. “Sweetness can hurt the stomach”) then “sweetness” becomes a part of “drink” (as do all other flavors, for symmetry).

3.4 Relational and Set-Theoretic Features

Many utterances in natural language perform actions akin to relational database operations, and use set-theoretic operators. For example, consider the following utterance (taken from sentence (5) of the interaction) and an interpretation for it.

`The bartender makes a random sweet drink from the menu.`

```
bartender.make(random.choice(filter(lambda drink:
'sweet' in drink.properties, menu.drinks)))
```

The phrase “a random sweet drink” is really a *dynamic reference*, because it points not to a static object, but rather, gives a relational specification for it (i.e. drink, random, sweet), which in turn implies a procedure to pick from a “database” of objects (i.e. find the menu, then find the drinks, then filter out only the sweet drinks, and pick one at random).

In addition, English is imbued with set-theoretic features such as the comparative (e.g. “longer,”) and the superlative (e.g. “longest,”) adjectives and adverbs; as well as set-facilitation determiners (e.g. “*all* drinks have”, “*some* drinks … while *other* drinks”). A comparative allows a choice between a set of size two (e.g. “the cheaper drink”), and a superlative among a set of any size. The criteria for comparison can either be contained in the semantics of the word itself (e.g. “cheapest”) or can refer to some procedure usually contained in a complementizer phrase (e.g. “the drink *which Bill would like best*”). Set-facilitation determiners facilitate LISP-style processing of elements (e.g. “order *all* the sweet drinks”,), however, there are almost ambiguous ways to cut up sets (e.g. “*most* of the drinks are sweet”) which remain ambiguous, or may require additional background knowledge to disambiguate.

It is far more common to find set-manipulation procedures (e.g. map, filter, reduce, max, min) implied in natural language that it is to find explicit looping language, and in fact such procedures are composed quite elegantly in natural language, e.g.:

`The customer buys all the sweet drinks under $2.`

```
map(customer.buy,
     (filter sweet_drink: sweet_drink.price < 2,
     filter(lambda drink: 'sweet' in drink.properties,
     menu.drinks))
```

In a study of non-programmer’s solutions to programming problems, Pane, Ratanamahatana & Myers (2001) report that people tend not to speak of iteration or looping in explicit terms. Perhaps one explanation for why this is lies in the observation that most set-manipulation procedures (e.g. “the cheapest drink on the menu”) do not demand to be evaluated immediately; rather, they are subject to *lazy evaluation*, at a future time, only if needed. In contrast, explicit looping language (e.g. “look at each drink and price on the menu, if the price is lower than any seen thus far, remember that drink” etc) would force the procedure to be attended to immediately, which would occupy valuable space in human short-term memory and the human discourse stack (which some have reported has only a practical depth of 2).

3.5 Narrative Stances and Styles

One of the affordances of stories is that they can be told through a choice of narrative stances, such as first-person, third-person, and mixed-person playwright. While it can be argued that varied styles are also present in programming languages, such as object-oriented, functional, and imperative, the difference is that narrative stances often can map to the same programmatic representation (making some basic inferences), while each style of programming has essentially married itself to a set of difficult-to-reverse representational choices.

Because different narrative stances can be used somewhat interchangeably, a story with many different narrators using different stances (or the same narrator switching between stances when convenient) could still plausibly be coherent; the same probably cannot be said of different programmers augmenting the same code with different paradigms for programming.

What grants the first-person, third-person, and mixed-person playwright stances equivalence is the deixis (meaning, contextualized pointer) of words like “I,” “him,” “here,” and “that.” Consider the following utterances:

a) I want to make a bar with a customer. (1st p. programmer)

b) There is a customer in the bar. (3rd p. narrator)
c) I am a customer sitting on a stool. (1st p. customer)
d) The bartender said, "Here is a customer" (mixed person playwright)

During interpretation of a story, a deictic stack is constantly updated, maintaining the current stance or point-of-view of the speaker, and dynamically mapping words like “I,” “him,” and “here” to their appropriate referents. In a), “I” is the programmer, so the action “make” is the same as declaring “bar” as an object with the part “customer.” In b), “there” is an existential declaration. In c), someone is speaking who declares that he is a customer, and by virtue of his current location on a stool (indicated by the progressive form, “am sitt*ing*”), and that the stool is inside the bar, we can infer by spatial transitivity that the customer is inside the bar. Finally in d), the narrator’s utterance of “the bartender said” allows us to set the speaker “I” to “bartender;” consequent from that, because the bartender is “inside the bar,” the other deictic mappings are updated too; thus, the utterance “Here is a customer” is resolved to “<inside bar> is a customer.”

3.6 Prototypes and Background Semantics

Natural language relies heavily on background knowledge and the manipulation or augmentation of known prototypes. Lakoff and Johnson (1980) have long suggested that language is inherently metaphorical, always building upon the scaffolding of existing knowledge. They assert that mathematics and physics are unintuitive precisely when they do not map metaphorically into our existing understanding.

If a programmatic interpreter of natural language hopes to be successful, it should find a way to establish some background knowledge. The most basic prototype object in a story is none other than *a person*; ordinary people have a great deal of knowledge regarding sociality and naïve psychology, and much of this is meant to assist us in social situations, but it is also crucial in understanding characters in stories. To take the Section 2 extended interaction for example, to start, we should be able to recognize that “bartender” and “customer” subtype “person,” and that “bar” subtypes “object” or “furniture,” etc. Knowing some of the typical abilities of all people (such as “eating” “drinking”) can help the interpreter infer abductively from the sentence, “Foo drinks something” that Foo is likely a “person” (unless we are in the realm of animals!). And it is not only objects which have prototypes defined by background semantics; functions *i.e.* verbs also have prototypes, and in fact, it is possible to organize verbs along hierarchies and semantic alternations classes. Levin's Verb Classes (1993) suggests one possible organization.

Until recently, there was a paucity of publicly-available resources which could supply the type of background semantics. But ConceptNet (Liu & Singh, 2004b), a large semantic network of common sense knowledge derived from MIT's Open Mind Common Sense project, is beginning to fill this role. ConceptNet's relational ontology suggests it could map easily into object-oriented structures. A sampling of possible mappings is given below:

```
CapableOf(x,y) → x.y(); LocationOf(x,y) → y.x
PropertyOf(x,y) → x.y;  PartOf(x,y) → x.y
IsA(x,y) → class x(y);  EffectOf(w.x,y.z) → w.x(): y.z
```

There are many practical ways in which this background knowledge could support programmatic interpretation. For example, knowing that `CapableOf("bin", "hold things")`, we can more confidently interpret "bin" as a list or a container object. Knowing some examples of fruits, if we encounter an "apple martini" and a "pear martini," we can generalize this to "fruit martini" where the fruit is a parameter.

4. METAFOR'S IMPLEMENTATION

4.1 Important Components

Parser – Uses the MontyLingua natural language understanding system (Liu, 2004a) to first perform a surface parse of each sentence into VSOO (verb-subject-object-object) form.

Programmatic Interpreter – First a small society of semantic recognizers mulls over the VSOO syntactic parse to identify existing objects in the code, special structures (like scoping statements, lists, quotes, if-then structure), and objects for which there exists some commonsense type information (e.g. common agents, color names, flavors, etc); second, a set of understanding demons, each capable of mapping a VSOO structure to some action or change in the code model, is run over the parsed sentences; third, the interpreter has a state tracker which maintains a deictic discourse stack, the current scope, and the current interpretive context (i.e. declarative versus procedural, as explained in Section 2) which are used by the understanding demons.

MetaforLingua -- This is the underlying knowledge representation of the code model; it is worth mentioning as a component in its own right because it is self-maintaining in that it is responsible for updating its own representation. As introduced in Section 3, all objects outside of if-then structures, have the form:

```
(full_name, arguments, body)
```

As the contents of the arguments and the body changes, the dynamic type inspector demon will assign it a different type. There is also a symmetry inspector which can propagate the implications of any change. For example, suppose that a "drink" had a part called "`possible_flavors = ['sour','sweet']`." Upon arriving at the statement "A drink's sweetness hurts the stomach," the dynamic type inspector promotes "sweet" to an object called "sweetness" with the constituent function "hurt(stomach)." Then the symmetry inspector propagates the change to promote the sister atom "sour" to an object called "sourness," etc.

Code Renderer – Currently, only a code renderer for Python exists, but because MetaforLingua is rather generic, renderers for other languages like LISP and Java could easily for written.

Introspection – An introspection feature allows any code object or function to explain itself using generated story language when that code object is moused over. This is more or less the inverse of mapping stories to code, but can be particularly useful to a novice who has difficulty reading code.

Dialog – The system agent generates natural language dialog to relate to the user how it has interpreted her utterances, in order to offer the user transparent access to the system interpreter. The goal of dialog is also to communicate any system confusion about an ambiguous utterance. See Figure 1 for a sample dialog.

User Interface – Ideally we would like to integrate a graphical visualization of the code model, or better yet, for a domain such as a MUD or MOO, the window could contain a real-time simulation of the MUD/MOO as it is updated.

4.2 Simplifying Assumptions

Metafor implements a good deal of the theoretical suggestions given in Section 3, but it also makes some simplifying assumptions in light of the realities of natural language understanding tools. However, we suggest that although a person using Metafor for brainstorming might initially have to get accustomed to the interpretive capabilities of the system (for example, choosing a meaningful verb is quite important), in further interactions a person should not feel that his or her storytelling abilities and freedoms are significantly impaired by these restrictions.

First, we assume that functions will always correspond to verbs. While this is generally true of English, there are exceptions. For example, in the sentence "When the drink is available," the important predicate is not "be" but "be available." In Metafor, there is a heuristic solution to this: if a verb itself is semantically too generic (e.g. "be", "get") then the whole verb phrase is made the function name, so "drink is available" would parse to "be_available(drink)." Also, many verbs actually span more than one word; these are idioms called *phrasal verbs* (e.g. "wake up," "put away") and they are handled to a limited extent by the natural language parser.

Second, we have to make assumptions about scoping expressions. Consider the scoping expression below (taken from Section 2, sentence (5)):

```
When the customer asks the bartender to choose, the bartender makes a random sweet drink from the menu if the customer's age is under 30; or else the bartender makes a sidecar.
```

Here, an if-then-else expression is nested within the function `ask_bartender_to_choose()`, because the utterance "when the customer ask the bartender to choose" is a scoping expression. However, there is ambiguity as to whether the next sentence still falls inside that function's scope, or if we have returned to the global scope. To address this ambiguity, we make the assumption that the scope will always escape back to global following a sentence break, *unless* a connector phrase (e.g. "then," "next," "afterwards") or sequencing phrase (e.g. "first .. second ..") is used. There is a sense that this corresponds accurately to the role of connector and sequencing phrases in actual human communication, but admittedly, humans also have an abundance of commonsense knowledge to support each decision about scope.

A third simplifying assumption is made about function argument structure. In natural language, each verb implies a very sophisticated semantics for the arguments which follow it; however, when we map verbs and their arguments in a literal fashion to function-argument structure, much is lost. Consider this example from Section 2 sentence (4): The utterance "give the drink to the customer" maps literally into the function-argument: "`give(drink,to_customer=customer);`" however, what is lost in this simplification are the additional semantics of the verb give. The event "give" should have three major thematic roles for "giver," "recipient," and "object_given," and giving something to someone also implies that they receive it. In our example, this would mean that "`give(drink,to_customer=customer)`" should actually trigger "`to_customer.receive( drink, from_person = bartender)`." However, the incorporation of sophisticated background knowledge into Metafor is yet to be implemented. NB, the study of *frame semantics* is precisely concerned with this task of cataloging verb-argument structure, as represented by Berkeley's FrameNet project (Baker, Fillmore & Lowe, 1998), and there is hope of incorporating such a resource in the near future.

5. USER STUDY

We conducted a 13-person user study to gain a better sense for the potential utility of Metafor's story-as-code visualization approach to both non-programmers and intermediate programmers. (We will include an updated study at press time; specifics here are indicative only).

Volunteers were MIT undergraduates, 7 of whom identified themselves as intermediate programmers, and 6 as non-programmers. The question being studied was: *"How does brainstorming code with Metafor affect a volunteer's self-assessment of the difficulty of a programming task?"* This is closer to testing the claim made in Section 1 about brainstorming benefiting intermediate programmers. Perhaps strangely, we asked even non-programmers how long it might take them to complete a programming task, even though they may not have the capability to accurately assess this. In light of this, we present the results as ratios of time rather than absolute times.

Each volunteer was given the same description of a programming task to program the basic high-level behaviors (excluding GUI, timing, etc) of the Pacman game. An assessment of how long the task would take them was elicited from each volunteer (baseline #1). Then, each volunteer was asked to spend two minutes writing a short story about Pacman, and then once again asked for a time-to-complete-task estimate (baseline #2). Finally, the examiner spent 5 minutes with each volunteer in front of Metafor, typing in their story into Metafor. The examiner who is familiar with the grammatical limitations of the Metafor interpreter normalized some phrasing of the volunteer's sentences, but the volunteer was asked to object if the examiner ever introduced new information into the description. At the conclusion of the Metafor interaction, each volunteer was asked a final time for a time-to-complete-task estimate. Volunteers were also asked how likely they would be to adopt brainstorming-on-paper and brainstorming-with-Metafor on a Likert5 scale (5=very likely, 1=very unlikely).

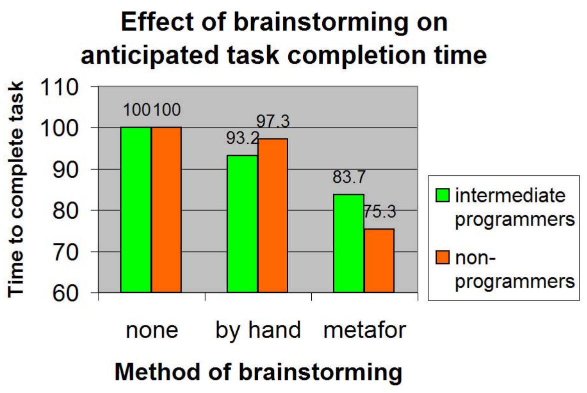


Figure 2: The effect of brainstorming on each volunteer's self-assessment of time-to-complete Pacman task. Times were normalized to 100 to allow comparison.

With regard to self-assessed time-to-complete the Pacman task, both non-programmers and intermediate programmers reported that prior brainstorming with Metafor had a great positive impact that brainstorming by hand. In general, non-programmers felt that brainstorming by hand didn't bring them much closer to completing the task, and as one volunteer said, "I still wouldn't know how to program it." Metafor made a more substantial positive difference to non-programmers than for intermediate programmers, but both programmers and non-programmers remarked that the system was "cool," and was eager to work with it. With regard to the results for the "adoption" question shown in Figure 3, both groups seemed unlikely to possibly about to adopt brainstorming by hand. One intermediate programmer felt that writing something down did not make progress in actualizing the code. Both group liked Metafor better, particularly because "it seems like it would be cool to play with," and as another respondent said "I think it would be a fun way to draft a project." Three respondents were surprised that their stories translated so directly to programs and one said he would write the story differently knowing now how the computer processes the text.

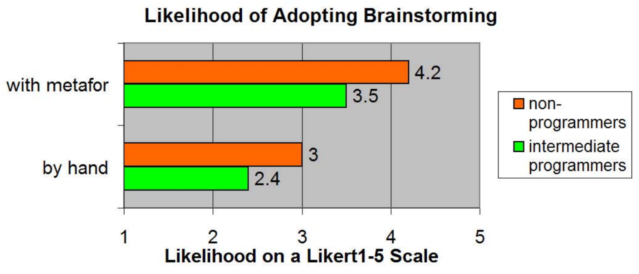


Figure 3: The likelihood of adopting Metafor as a brainstorming step, compared against brainstorming by hand.

We are encouraged by these initial findings. What was particularly striking were the non-programmers' enthusiasm. It was also interesting that, after working with Metafor, several respondents expressed regret that they did not express their story differently; to us this suggests that after having viewed a programmatic interpretation of natural language, a user is gaining valuable reflection about her own storytelling tendencies. The important thing is that the process of learning is entertaining.

6. RELATED WORK

Since Metafor is meant as an aid to programming, it falls under the genre of *case-tools*. In the literature of case tools, two interfaces are worth mentioning. First, Tam, Maulsby, and Puerta developed a system called U-Tel (1998) which elicits a story about a task from a person, and allows the person to manually highlight and annotate words in the text with their possible roles in the code. Second, Hars & Marchewka developed a natural language case tool (1996) which maps expert-system rules, stated in English, into a yes/no decision flowchart whose nodes are large unparsed natural language utterances. Our approach differs from Tam *et al.*'s approach in that Metafor tries to automatically interpret the user's story. Non-programmers who do not understand the specifications of natural language may not be able to manually annotate code; also the fact that almost every utterance in a story has some actionable consequence in Metafor may bring some fundamental assumptions to the foreground which in a system like U-Tel might simply be passed over, or it may have the effect of encouraging more precise articulations to be made.

Metafor and Hars & Marchewka's system share a common goal of helping to automatically visualize a natural language story; however, Hars & Marchewka's system seems mainly capable of understanding if-then-else structure, whereas Metafor is capable of visualizing further structure.

7. CONCLUSION

Metafor is an intelligent user interface which encourages a person to *tell a story* about a made-up world, or to describe a task like Pacman in plain English. All the while, Metafor constantly tries to understand each utterance through a programmatic interpreter, whose implementation derives from a theory we are developing of the programmatic semantics of natural language. Metafor visualizes each story utterance as changes and additions to some Python code representation of the story.

To be sure, this code visualization is not complete; it is merely *scaffolding code* which reifies many of the user's high-level descriptions of behavior. Metafor's natural language parser and programmatic interpreter will not be able to correctly interpret any arbitrarily complex English utterance. However, English has a structural regularity (it is a SVO subject-verb-object language), Metafor will usually produce *some* interpretation. Metafor is very transparent about what it best understands and worst understands, as a system agent explains in plain English what it thought the user asked. We believe that a person could quickly adapt to producing utterances which would be well-understood by Metafor's interpreter, yet inherently this adaptation *does not limit what's thinkable* because it does not limit what can be said, it is only picky about phrasing.

We are encouraged by the results of a user study for Metafor, which showed that both non-programmers and intermediate programmers found the system to be an effective brainstorming and project planning tool, more so that simply writing a story down. One of the more striking reactions from some of the non-programmer participants was that Metafor was like a programming tutor which they might have unlimited access to. In the end, participants found that Metafor is fun, and *fun* cannot be emphasized enough; it provides programming novices a fun way to gain intuition for programming by allowing them to *tinker* with something which is responsive to anything they type. If we could make programming more like a toy that even younger kids could play with, our hearts can only imagine the educational implications.

8. REFERENCES

[1] Collin F. Baker, Charles J. Fillmore, John B. Lowe: 1998, The Berkeley FrameNet project. In Proceedings of the COLING-ACL, Montreal, Canada.

[2] A. Hars, J.T. Marchewka: 1996, Eliciting and mapping business rules to IS design: Introducing a natural language CASE tool. In: Ebert, R.J; Franz, L.: 1996 Proceedings Decision Sciences Institute, Vol.2, pp. 533-535.

[3] George Lakoff, Mark Johnson: 1980, *Metaphors We Live by*. University of Chicago Press.

[4] Beth Levin: 1993, *English Verb Classes And Alternations: A Preliminary Investigation.* The University of Chicago Press.

[5] Henry Lieberman and Hugo Liu: 2004a, Feasibility Studies for Programming in Natural Language. In *Lieberman, Paterno & Wulf (Eds.) End-User Development*. Kluwer.

[6] Hugo Liu: 2004a, MontyLingua v2.1 Free Natural Language Understanding Toolkit and API available at: http://web.media.mit.edu/~hugo/montylingua/

[7] Hugo Liu and Henry Lieberman: 2004b, Toward a Programmatic Semantics of Natural Language. *Proceedings of the 20th IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society Press.

[8] Hugo Liu and Push Singh: 2004b, ConceptNet: A Practical Commonsense Reasoning Toolkit. *BT Technology Journal 22(4)*. Kluwer.

[9] Erik T. Mueller: 1999: Prospects for in-depth story understanding by computer.

http://arxiv.org/html/cs.AI/0003003.

[10] J.F. Pane, C.A. Ratanamahatana, & B.A. Myers: 2001, Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies, 54(2)*, 237-264.

[11] R.C. Tam, D. Maulsby, and A.R. Puerta: 1998, U-TEL: A Tool for Eliciting User Task Models from Domain Experts. *Proceedings of IUI'98*, pp. 77-80.