# 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. So-called *programming languages,* while easy for the computer to accurately convert into code, are, unfortunately, difficult for people to write and understand.

We explore the idea of using descriptions in a natural language as a representation for programs. While we cannot yet convert arbitrary English to fully specified code, we can use a reasonably expressive subset of English as a *visualization tool*. Simple descriptions of program objects and their behavior generate *scaffolding* (underspecified) code fragments, that can be used as feedback for the designer. Roughly speaking, noun phrases can be interpreted as program objects; verbs can be functions, adjectives can be properties. A surprising amount of what we call *programmatic semantics* can be inferred from linguistic structure. 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
We have developed an intelligent user interface, Metafor, for visualizing a person's interactively typed stories as code. As a person types a story into Metafor, the system continuously updates a side-by-side "visualization" of the person's narrative as *scaffolding code.* This code may not be directly executable, but it is meant to help a person reify her thoughts. We believe that Metafor is a novel system which can accomplish at least two main *goals*: 1) assist novice programmers in developing intuitions about programming; and 2) facilitate intermediate programmers with a brainstorming and "outlining" tool, ahead of "writing."


*IUI'05*, January 10–13, 2005, San Diego, California, USA.
ACM 1-58113-894-6/05/0001.

### 1.1 Scope and Limitations
Our parser cannot understand every grammatically legal construction. And, although our parser does use a large knowledge base of common sense knowledge, it doesn't know everything a programmer might think of saying. 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.

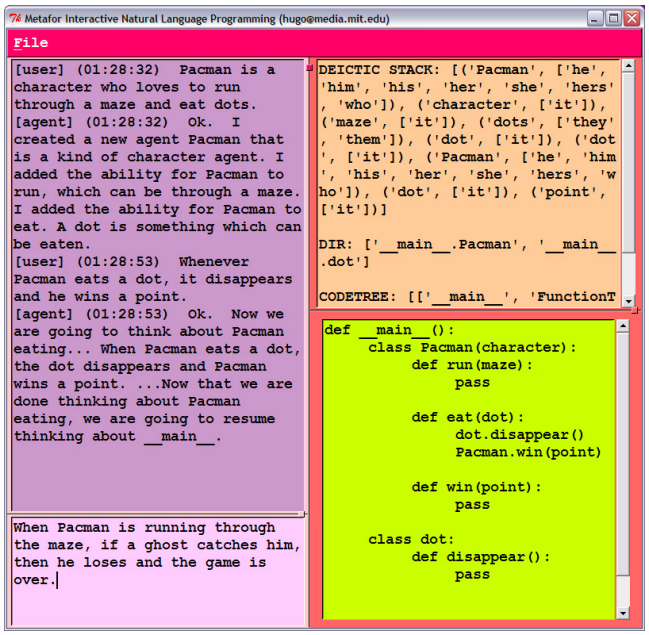


**Figure 1.** A screenshot of Metafor. Clockwise from the lower left corner, the four windows display 1) the narrative being entered; 2) an interaction log; 3) Metafor's representation of the parse tree (for demo and debugging purposes only: not shown to beginning users); and 4) the visualization code, in Python.

We are encouraged by experience with MOOs (Multi-user dungeons Object-Oriented) and other text-based interaction games (readers unfamiliar with MOOs may wish to consult (Bruckman, 1997)), that achieve usable interaction even with very simple template-based parsers; with natural language interfaces to databases and search engines; and with conversational "chatbots".

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. An introspection facility allows a user to see, for a given piece of code, what natural language expressions can be generated from it.

## AN EXAMPLE INTERACTION

Below we run through a brief scenario of actual Python visualization code outputted by interacting with the Metafor program.

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

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

This sentence is actually quite rich in information. Metafor's interpreter infers several facts from parsing this sentence: "There is a bar," "A bar has a part called bartender," "there is a bartender" (implied), and "A bartender can make drinks." Note that the "who" in (1) is resolved as "the bartender," not as "the bar" because only "bartender" is animate (some limited semantic knowledge is used to determine 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) 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 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 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 behavior is to treat the main verb as the predicate/function.

## 2. AMBIGUITY IS YOUR FRIEND

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". How should this object be represented and what should be parameterized? 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. METAFOR'S IMPLEMENTATION

### 3.1 Important Components

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

**Integration with Common Sense Knowledge** – One of the major distinguishing features of MontyLingua is its integration with a large knowledge base of Common Sense Knowledge, ConceptNet (Liu and Singh, 2004), derived from Open Mind, a corpus of 750,000 natural language statements of Common Sense knowledge contributed by 15,000 Web community volunteers. We have already mentioned how this helps us dereference anaphora. But it is also essential in constraining the proliferation of grammatically possible but intuitively implausible parses that might be generated by more conventional parsers.

**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) 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. 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 be 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 MOO, the window could contain a real-time simulation of the MOO as it is updated.

# 4. USER STUDY

We present only a brief overview of a 13-person study with both non-programmers and intermediate programmers, asking: *"How does brainstorming code with Metafor affect a volunteer's self-assessment of the difficulty of a programming task?"* Details will appear in a forthcoming paper.

We could not directly test task performance since Metafor is only intended as a brainstorming tool (imagine testing an outliner for prose writing). We asked volunteers to estimate the time for programming a simple Pacman game, write a short story describing the game with Metafor, then re-estimate task completion time to see if Metafor added value. We also asked about how likely they were to brainstorm with Metafor versus paper. The examiner would occasionally gently rephrase the volunteer's sentences if the volunteer strayed too far outside Metafor's syntactic competence; this was not a test of the coverage of Metafor's grammar.

Non-programmers estimated that Metafor reduced task time by 22%, the figure was 11% for intermediate programmers. Non-programmers said that they were halfway between "likely" and "very likely" to adopt Metafor for brainstorming, whereas they only rated themselves well below "likely" to use paper brainstorming. Intermediate programmers were more confident of their paper brainstorming skills, but nevertheless said they were 31% more likely to choose Metafor over paper methods.

## 4.1 Related Work

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. U-Tel also does not produce code directly, but input to a model-based UIMS where the interface can be further specified. Hars & Marchewka's natural language case tool (1996) maps expert-system rules, stated in English, into a yes/no decision flowchart whose nodes are large unparsed natural language utterances.

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, based on a study performed by Pane *et al.* (2001). In that paper, we present some mixed-initiative dialogs that can mitigate some of the ambiguity and underspecification problems associated with natural language descriptions. We have also been exploring how natural language might inherently be interpretable under a programmatic semantics framework (Liu & Lieberman, 2004b).

# 5. CONCLUSION

The idea of basing a programming language upon natural language dates back to the earliest days of high-level programming languages. COBOL was an attempt to make programming code as similar as possible to English, in contrast to FORTRAN's metaphor of mathematical formulae. The hope was to make programming accessible to non-technical business users, But computer science has long since abandoned the project in the face of difficulties of parsing and knowledge representation. We show that modern parsing techniques and integration of Common Sense knowledge can, if not enable full programming, serve as a bridge between people's intuitive narrative capacities and conventional programming. And, most importantly, we hope interfaces like Metafor can put some of the *fun* back into programming.

# 6. REFERENCES


[1] Amy Bruckman: 1997, *MOOSE Crossing.* PhD Thesis, MIT.

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

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

[5] 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.

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

[7] 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.

[8] 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.