Toward a Programmatic Semantics of Natural Language

Hugo Liu and Henry Lieberman
MIT Media Laboratory, Massachusetts Institute of Technology
{hugo, lieber}@media.mit.edu

Abstract

Natural language is imbued with a rich semantics but unfortunately its complex elegance is often mistaken for mere imprecision. Because complete parsers of English are not yet achievable, people assume that it is not feasible to use English directly as a means of instructing computers. However, in this paper, we show that English descriptions of procedures often contain programmatic semantics *– linguistic features that can be easily mapped into programming language constructs. Some linguistic features can even inspire new ways of thinking about specifying programs. Far from being hopelessly ambiguous, natural languages exhibit important principles of communication that could be used to make human-computer communication more natural.*

1. English as a Programming Language

Natural human language is often dismissed as being too informal and ambiguous to compute with and program in because it does not obey the rigor of logic. Rather than relying on absolute truth and deduction, natural language and human reasoning rely on abduction, or evidentiary reasoning. By modeling abduction probabilistically, it may be possible then, to create quasi-formalisms for natural language.

Recently, we investigated the feasibility of enabling programming in natural language [1] by reviewing Pane and Myer's study of fifth graders' plain English descriptions of a Pacman game [3]. We then implemented a system called Metafor, capable of rendering simple children stories like "Humpty Dumpty" as "scaffolding code" (descriptive, but not complete enough to execute) in Python. The surprising conclusion of these exercises was that in most cases, fairly direct mappings are possible from parsed English to the control and data structures of traditional high-level programming languages like LISP and Python, suggesting that even a naïve, direct *transliteration* of English into programmatic code is unexpectedly good. If mapping difficulties can be attributed to anything, it is that English often exhibits some very desirable programming semantics that are either not yet available in programming languages, or have only recently become available as advanced features. Drawing from the findings of our feasibility study and implementation of Metafor, this paper aims to elucidate some of the basic programmatic features of natural language with English as the exemplar.

2. Programmatic Features of English

In this section, we rely on salient examples to illustrate the mappings from natural language syntax and semantics to programming constructs. Examples are drawn from the Pacman domain, given in [3].

Syntactic Typing. For the most part, the major syntactic parts-of-speech of natural language correspond to distinct syntactic types in programming languages. Action, or non-copular, verbs (everything except verbs like *to be*, and *to seem*) map to functions (*e.g.* "eat", "chase"), while noun phrases map to classes (*e.g.* "the maze", "dots"). Adjectival modifiers map to properties of a class (*e.g.* "yellow blinking dots"), and adverbial modifiers map to auxiliary arguments to functions (*e.g.* "chomps dots quickly" ⇔ chomp(dot,speed=quickly)). A class can be further distinguished into active agents imbued with methods, or passive structures with only properties. A noun-phrase that is a *subject* to an action verb is usually an agent (*e.g.* "Ghosts chase"). Conversely, if it only plays object to an action verb (*e.g.* "eats dots") or subject to a copular-passive verb (*e.g.* "dots are eaten"), this is more suggestive of a passive structure. Prepositional phrases attached to a verb usually play the role of function argument (*e.g.* "eats dot by chomping"), and the nuanceful ways that different prepositions imply different sorts of arguments (*e.g.* governing dimensions of duration, speed, manner, etc.) are well beyond the scope of this paper.

Inheritance. Natural language relies heavily on inheritance from known exemplars. In fact, we can view the primary purpose of having massive amounts of "common sense" knowledge about the everyday world as building up a sufficiently rich programmatic library of exemplars from which we can inherit during story understanding. Linking a novel concept to known exemplars can be done explicitly, but also implicitly via structural abduction. First, explicit inheritance is

seen in *"is a"* statements (*e.g.* "Pacman is a character") and appositives (*e.g.* "Pacman, the main character, eats dots"). Second, it is possible to insinuate an inheritance via analogy or abductive hypothesis by imbuing the novel concept with some of the property and action structures of an exemplar (*e.g.* "Pacman eats dots and runs around" ➔ Pacman is possibly a creature of some sort). These implied inheritances are computable via structure mapping. The need for large libraries of commonsense exemplars is starting to be addressed by resources like ConceptNet [3] (www.conceptnet.org), which has at present, a million assertions about the everyday world, and is also capable of defining dynamic context, and making structural analogy.

Reference. Whereas in most programming languages there is a uniform convention for referring to an object, natural language is characterized by a greater diversity of reference mechanisms for which we propose four general kinds: dominion, set selection, metonymy, and anaphora. First, dominion notation uses the possessive marker "*'s"* and the possession preposition "*of"* to establish a scope path much in the same way as OOP's dotted namespace notation (*e.g.* "Pacman's score" ⇔ Pacman.score; "the width of the dot" ⇔ dot.width). Second, in cases where it is necessary to distinguish a particular object from within a set of objects of the same class, distinguishing features of the desired object are used to select from that set (*e.g.* "the blinking dots"⇔filter(lambda dot: dot.blinking, dots); "the dot in front of Pacman" ⇔filter(lambda dot: in_front_of(dot.pos,Pacman.pos), dots)). Third, objects may be referred to via a surrogate metonym, and the choice of metonym can either foreshadow a particular role to be played by that object (*e.g.* "Pacman's *food*" (rather than *dot*)), or be used to highlight certain semantical aspects of an object to be preferred in the ensuing interpretation (*e.g.* "The hungry protagonist" rather than *Pacman*). Fourth, anaphora (*e.g.* it, she, this) can be thought of as a restricted kind of metonymic reference which relies heavily on knowledge of entities in the current discourse context. Programmatically, we can conceptualize the resolution of anaphora as structural-abductive selection from entities in the current namespace (*e.g.* In "Pacman eats dots. He is hungry," the pronoun *he* refers to Pacman rather than to dots, because an agent has hunger, but not a passive object.)

Conditionals, Iteration, and Stepping. Pane and Myers concluded in their studies of natural programming that people tend not to speak of iteration or looping in explicit terms [3]. Since the human discourse stack is relatively small (some have proposed n=2), the complex and nested nature of iterations and loops are usually flattened out into double-clause structures. (That so much information can be flattened out is testimony to the rich semantics of natural language.) In fact, conditionals, iteration, and stepping all share this double-clause property. First, natural language conditional expressions often assume a binary antecedent-consequent structure such as subjunctive form (*e.g.* "A dot would disappear if Pacman ate it"), or an independent clause augmented with a "when" dependent clause (*e.g.* "When a dot is eaten, it disappears and Pacman scores a point"). Second, iterations also share this double-clause structure. An iteration can be implied by a continuation progressive verb phrase augmented with a "terminate-when" or "until" dependent clause which acts as a termination condition (*e.g.* "Pacman keeps eating dots and he finishes when all the dots have been eaten"). To step through a list of tasks, either an ordinal numbering is given (*e.g.* first, second), or more informally, double-clause pairwise orderings are given (*e.g.* "Do W, then do X. After X, do Y, followed by Z."), forming what resembles a linked list.

Functional Composition, Genericity, and Lazy Evaluation. The specialization of parts-of-speech, and prevalence of junction constructions (*e.g.* and, or, but, when, if*,* which*)* afford natural language the capability for elegant and seemingly flat functional compositions (*e.g.* consider that it takes at least 4 map/filter iterations to even naively represent: "Each ghost follows his own shortest path to Pacman"). Natural language is also generic enough to use the same syntax to declare and compute variables, a manner similar to *generic functions* of the Common LISP Object System (*e.g.* "Pacman eats yellow dots" can depending on what's known, declare that dots are yellow, or apply "eat" only to the subset of dots which are yellow). Unlike in most programming languages, the economic and goal-driven nature of story understanding causes evaluation of natural language expressions to almost always be lazy. For example, it may be sufficient to acknowledge that a procedure for generating the "shortest path" exists without actually specifying one.

3. References

[1] Lieberman H. & Liu, H. (2004). Feasibility Studies for Programming in Natural Language. In *Lieberman et al. (Eds.) Perspectives in End-User Development.* Kluwer.

[2] Liu, H. & Singh, P. (2004). Commonsense Reasoning in and over Natural Language. *KES'2004.*

[3] Pane, J.F. & Myers, B.A. (1996). Usability Issues in the Design of Novice Programming Systems, CMU Technical Report CMU-CS-96-132. Pittsburg, PA.