NLP (Natural Language Processing) for NLP (Natural Language Programming)

Rada Mihalcea[1], Hugo Liu[2], and Henry Lieberman[2]

[1] Computer Science Department, University of North Texas
rada@cs.unt.edu
[2] Media Arts and Sciences, Massachusetts Institute of Technology
{hugo, henry}@media.mit.edu

Abstract. Natural Language Processing holds great promise for making computer interfaces that are easier to use for people, since people will (hopefully) be able to talk to the computer in their own language, rather than learn a specialized language of computer commands. For programming, however, the necessity of a formal programming language for communicating with a computer has always been taken for granted. We would like to challenge this assumption. We believe that modern Natural Language Processing techniques can make possible the use of natural language to (at least partially) express programming ideas, thus drastically increasing the accessibility of programming to non-expert users. To demonstrate the feasibility of Natural Language Programming, this paper tackles what are perceived to be some of the hardest cases: steps and loops. We look at a corpus of English descriptions used as programming assignments, and develop some techniques for mapping linguistic constructs onto program structures, which we refer to as programmatic semantics.

1 Introduction

Natural Language Processing and Programming Languages are both established areas in the field of Computer Science, each of them with a long research tradition. Although they are both centered around a common theme – "languages" – over the years, there has been only little interaction (if any) between them[1]. This paper tries to address this gap by proposing a system that attempts to convert natural language text into computer programs. While we overview the features of a natural language programming system that attempts to tackle both the descriptive and procedural programming paradigms, in this paper we focus on the aspects related to procedural programming. Starting with an English text, we show how a natural language programming system can automatically identify steps, loops, and comments, and convert them into a program *skeleton* that can be used as a starting point for writing a computer program, expected to be particularly useful for those who begin learning how to program.

We start by overviewing the main features of a descriptive natural language programming system METAFOR introduced in recent related work [6]. We then describe in

[1] Here, the obvious use of programming languages for coding natural language processing systems is not considered as a "meaningful" interaction.

A. Gelbukh (Ed.): CICLing 2006, LNCS 3878, pp. 319–330, 2006.
© Springer-Verlag Berlin Heidelberg 2006

detail the main components of a procedural programming system as introduced in this paper. We show how some of the most difficult aspects of procedural programming, namely steps and loops, can be handled effectively using techniques that map natural language onto program structures. We demonstrate the applicability of this approach on a set of programming assignments automatically mined from the Web.

2 Background

Early work in natural language programming was rather ambitious, targeting the generation of complete computer programs that would compile and run. For instance, the "NLC" prototype [1] aimed at creating a natural language interface for processing data stored in arrays and matrices, with the ability of handling low level operations such as the transformation of numbers into type declarations as e.g. `float-constant(2.0)`, or turning natural language statements like *add y1 to y2* into the programmatic expression `y1 + y2`. These first attempts triggered the criticism of the community [3], and eventually discouraged subsequent research on this topic.

More recently, however, researchers have started to look again at the problem of natural language programming, but this time with more realistic expectations, and with a different, much larger pool of resources (e.g. broad spectrum commonsense knowledge [9], the Web) and a suite of significantly advanced publicly available natural language processing tools.

For instance, Pane & Myers [8] conducted a series of studies with non-programming fifth grade users, and identified some of the programming models implied by the users' natural language descriptions. In a similar vein, Lieberman & Liu [5] have conducted a feasibility study and showed how a partial understanding of a text, coupled with a dialogue with the user, can help non-expert users make their intentions more precise when designing a computer program. Their study resulted in a system called METAFOR [6], [7], able to translate natural language statements into class descriptions with the associated objects and methods.

Another closely related area that received a fair bit of attention in recent years is the construction of natural language interfaces to databases, which allows users to query structured data using natural language questions. For instance, the system described in [4], or previous versions of it as described in [10], implements rules for mapping natural to "formal" languages using syntactic and semantic parsing of the input text. The system was successfully applied to the automatic translation of natural language text into RoboCup coach language [4], or into queries that can be posed against a database of U.S. geography or job announcements [10].

3 Descriptive Natural Language Programming

When storytellers speak fairy tales, they first describe the fantasy world – its characters, places, and situations – and then relate how events unfold in this world. Programming, resembling storytelling, can likewise be distinguished into the complementary tasks of *description* and *proceduralization*. While this paper tackles primarily the basics of

building procedures out of steps and loops, it would be fruitful to also contextualize procedural rendition by discussing the architecture of the descriptive world that procedures animate.

Among the various paradigms for computer programming – such as logical, declarative, procedural, functional, object-oriented, and agent-oriented – the object-oriented and agent-oriented formats most closely embody human storytelling intuition. Consider the task of programming a MUD[2] world by natural language description, and the sentence *There is a bar with a bartender who makes drinks* [6]. Here, `bar` is an instance of the object `class bar`, and `bartender` is an instance of the agent (a class with methods) `class bartender`, with the capability `makeDrink(drink)`. Generalizing from this example, characters are reified as agent classes, things and places become object classes, and character capabilities become class methods.

A theory of programmatic semantics for descriptive natural language programming is presented in [7]; here, we overview its major features, and highlight some of the differences between descriptive and procedural rendition. These features are at the core of the Metafor [6] natural language programming system that can render code following the descriptive paradigm, starting with a natural language text.

3.1 Syntactic Correspondences

There are numerous syntactic correspondences between natural language and descriptive structures. Most of today's natural languages distinguish between various parts of speech that taggers such as Brill's [2] can parse – noun chunks are things, verbs are actions, adjectives are properties of things, adverbs are parameters of actions. Almost all natural languages are built atop the basic construction called *independent clause*, which at its heart has a *who-does-what* structure, or subject-verb-directObject-indirectObject (SVO) construction. Although the ordering of subject, verb, and objects differ across verb-initial (VSO and VOS, e.g. Tagalog), verb-medial (SVO, e.g. Thai and English), and verb-final languages (SOV, e.g., Japanese), these basic three ingredients are rather invariant across languages, corresponding to an encoding of agent-method and method-argument relationships. This kind of syntactic relationships can be easily recovered from the output of a syntactic parser, either supervised, if a treebank is available, or unsupervised for those languages for which manually parsed data does not exist. Note that the syntactic parser can also resolve other structural ambiguity problems such as prepositional attachment. Moreover, other ambiguity phenomena that are typically encountered in language, e.g. pronoun resolution, noun-modifier relationships, named entities, can be also tackled using current state-of-the-art natural language processing techniques, such as coreference tools, named entity annotators, and others.

Starting with an SVO structure, we can derive agent-method and method-argument constructions that form the basis of descriptive programming. Particular attention needs to be paid to the ISA type of constructions that indicate inheritance. For instance, the statement *Pacman is a character who ...* indicates a super-class `character` for the more specific class `Pacman`.

[2] A MUD (multi-user dungeon, dimension, or dialogue) is a multi-player computer game that combines elements of role-playing games, hack and slash style computer games, and social instant messaging chat rooms (definition from wikipedia.org).

3.2 Scoping Descriptions

Scoping descriptions allow conditional `if/then` rules to be inferred from natural language. Conditional sentences are explicit declarations of if/then rules, e.g. *When the customer orders a drink, make it*, or *Pacman runs away if ghosts approach.* Conditionals are also implied when uncertain voice is used, achieved through modals as in e.g. *Pacman may eat ghosts*, or adverbials like *sometimes* – although in the latter case the antecedent to the if/then is underspecified or omitted, as in *Sometimes Pacman runs away.*

```
package Customer;
sub orderDrink {
    my ($drink) = @_;
    $bartender = Bartender –> new(...);
    $bartender–>makeDrink($drink);
}
_____
package Main;
use Customer;
$customer = Customer–>new(...);
$customer–>orderDrink($drink);
```

```
package Customer;
sub orderDrink {
    my ($drink) = @_;
}
_____
package Main;
use Customer;
$customer = Customer–>new(...);
if ($customer–>orderDrink($drink)) {
    $bartender = Bartender –> new(...);
    $bartender–>makeDrink($drink);
}
```

Fig. 1. The descriptive and procedural representations for the conditional statement *When customer orders a drink, the bartender makes it*

An interesting interpretative choice must be made in the case of conditionals, as they can be rendered either descriptively as functional specifications, or procedurally as if/then constructions. For example, consider the utterance *When customer orders a drink, the bartender makes it.* It could be rendered descriptively as shown on the left of Figure 1, or it could be proceduralized as shown on the right of the same figure. Depending upon the surrounding discourse context of the utterance, or the desired representational orientation, one mode of rendering might be preferred over the other. For example, if the storyteller is in a descriptive mood and the preceding utterance was *there is a customer who orders drinks,* then most likely the descriptive rendition is more appropriate.

3.3 Set-Based Dynamic Reference

Set-based dynamic reference suggests that one way to interpret the rich descriptive semantics of compound noun phrases is to map them into mathematical sets and set-based operations. For example, consider the compound noun phrase *a random sweet drink from the menu.* Here, the head noun *drink* is being successively modified by *from the menu, sweet,* and *random.* One strategy in unraveling the utterance's programmatic implications is to view each modifier as a constraint filter over the set of all drink instances. Thus the object `aRandomSweetDrinkFromTheMenu` implies a procedure that creates a set of all drink instances, filters for just those listed in `theMenu`, filters for those having the property *sweet,* and then applies a random choice to the remaining drinks to select a single one. Set-based dynamic reference lends great conciseness and power to

natural language descriptions, but a caveat is that world semantic knowledge is often needed to fully exploit their semantic potential. Still, without such additional knowledge, several descriptive facts can be inferred from just the surface semantics of *a random sweet drink from the menu* – there are things called drinks, there are things called menus, drinks can be contained by menus, drinks can have the property *sweet,* drinks can have the property *random* or be selected randomly. Later in this paper, we harness the power of set-based dynamic reference to discover implied repetition and loops.

Occam's Razor would urge that code representation should be as simple as possible, and only complexified when necessary. In this spirit, we suggest that automatic programming systems should adopt the simplest code interpretation of a natural language description, and then complexify, or *dynamically refactor*, the code as necessary to accommodate further descriptions. For example, consider the following progression of descriptions and the *simplest common denominator* representation implied by all utterances *up to* that step.

```
a)    There is a bar. (atom)
b)    The bar contains two customers. (unimorphic list of type Customer)
c)    It also has a waiter. (unimorphic list of type Person)
d)    It has some stools. (polymorphic list)
e)    The bar opens and closes. (class / agent)
f)    The bar is a kind of store. (agent with inheritance)
g)    Some bars close at 6pm, others at 7pm. (forks into two subclasses)
```

Applying the semantic patterns of syntactic correspondence, representational equivalence, set-based dynamic reference, and scoping description to the interpretation of natural language description, object-oriented code skeletons can be produced. These description skeletons then serve as a code model which procedures can be built out of. Mixed-initiative dialog interaction between computer and storyteller can disambiguate difficult utterances, and the machine can also use dialog to help a storyteller describe particular objects or actions more thoroughly.

The Metafor natural language programming system [6] implementing the features highlighted in this section was evaluated in a user study, where 13 non-programmers and intermediate programmers estimated the usefulness of the system as a brainstorming tool. The non-programmers found that Metafor reduced their programming task time by 22%, while for intermediate programmers the figure was 11%. This result supports the initial intuition from [5] and [8] that natural language programming can be a useful tool, in particular for non-expert programmers.

It remains an open question whether Metafor will represent a stepping stone to real programming, or will lead to a new programming paradigm obviating the need for a formal programming language. Either way, we believe that Metafor can be useful as a tool in itself, even if it is yet to see which way it will lead.

4 Procedural Natural Language Programming

In procedural programming, a computer program is typically composed of sequences of action statements that indicate the operations to be performed on various data structures. Correspondingly, procedural natural language programming is targeting the generation

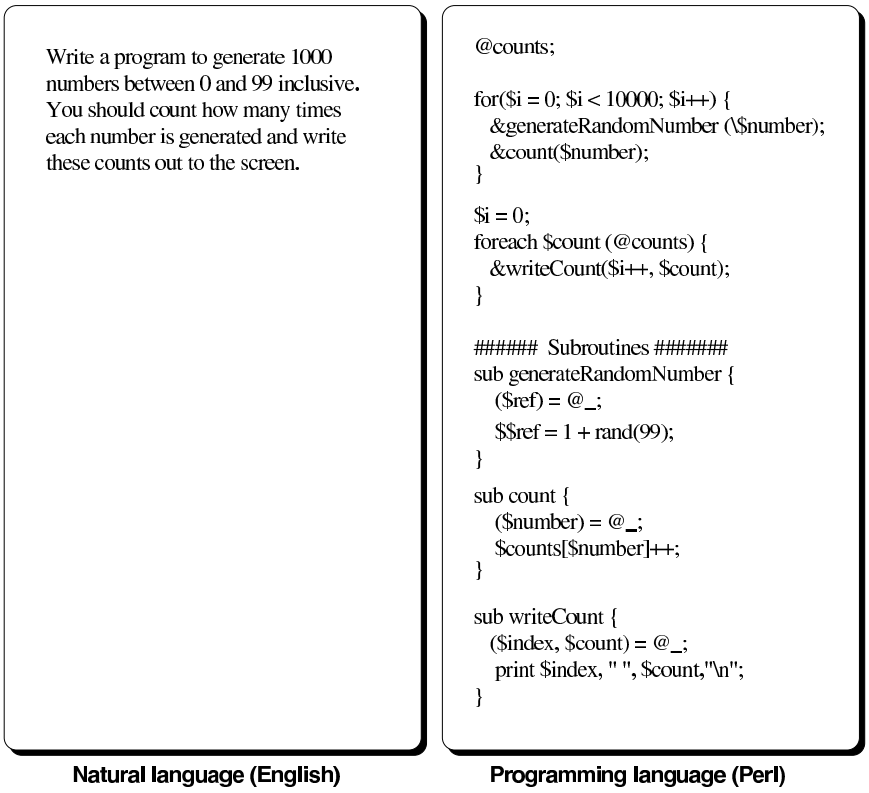

Fig. 2. Side by side: the natural language (English) and programming language (Perl) expressions for the same problem

of computer programs following the procedural paradigm, starting with a natural language text.

For example, starting with the natural language text on the left side of figure 2, we would ideally like to generate a computer program as the one shown on the right side of the figure[3]. While this is still a long term goal, in this section we show how we can automatically generate computer program skeletons that can be used as a starting point for creating procedural computer programs. Specifically, we focus on the description of three main components of a system for natural language procedural programming:

- The *step finder*, which has the role of identifying in a natural language text the action statements to be converted into programming language statements.
- The *loop finder*, which identifies the natural language structures that indicate repetition.
- Finally, the *comment identification* components, which identifies the descriptive statements that can be turned into program comments.

[3] Although the programming examples shown throughout this section are implemented using Perl, other programming languages could be used equally well.

Starting with a natural language text, the system is first analyzing the text with the goal of breaking it down into steps that will represent action statements in the output program. Next, each step is run through the comment identification component, which will mark the statements according to their descriptive role. Finally, for those steps that are not marked as comments, the system is checking if a step consists of a repetitive statement, in which case a loop statement is produced using the corresponding loop variable. The following sections provide details on each of these components (step finder, loop finder, comment identification), as well as a walk-through example illustrating the process of converting natural language texts into computer program skeletons.

4.1 The Step Finder

The role of this component is to read an input natural language text and break it down into steps that can be turned into programming statements. For instance, starting with the natural language text *You should count how many times each number is generated and write these counts out to the screen.* (see figure 2), two main steps should be identified: (1) [`count` how many times each number is generated], and (2) [`write` these counts out to the screen].

First, the text is pre-processed, i.e. tokenized and part-of-speech tagged using Brill's tagger [2]. Some language patterns specific to program descriptions are also identified at this stage, including phrases such as *write a program*, *create an applet*, etc., which are not necessarily intended as action statements to be included in a program, but rather as general directives given to the programmer.

Next, steps are identified as statements containing one verb in the active voice. We are therefore identifying all verbs that could be potentially turned into program functions, such as e.g. `read`, `write`, `count`. We attempt to find the boundaries of these steps: a new step will start either at the beginning of a new sentence, or whenever a new verb in the active voice is found (typically in a subordinate clause).

Finally, the *object* of each action is identified, consisting of the direct object of the active voice verb previously found, if such a direct object exists. We use a shallow parser to find the noun phrase that plays the role of a direct object, and then identify the head of this noun phrase as the *object* of the corresponding action.

The output of the step finder process is therefore a series of natural language statements that are likely to correspond to programming statements, each of them with their corresponding action that can be turned into a program function (as represented by the active voice verb), and the corresponding action object that can be turned into a function parameter (as represented by the direct object). As a convention, we use both the verb and the direct object to generate a function name. For example, the verb *write* with the parameter *number* will generate the function call `writeNumber(number)`.

4.2 The Loop Finder

An important property of any program statement is the number of times the statement should be executed. For instance, the requirement to *generate 10000 random numbers* (see figure 2), implies that the resulting action statement of [`generate` random numbers] should be repeated 10000 times.

The role of the loop finder component is to identify such natural language structures that indicate repetitive statements. The input to this process consists of steps, fed one at a time, from the series of steps identified by the step finder process, together with their corresponding actions and parameters. The output is an indication of whether the current action should be repeated or not, together with information about the loop variable and/or the number of times the action should be repeated.

First, we seek explicit markers of repetition, such as *each X*, *every X*, *all X*. If such a noun phrase is found, then we look for the head of the phrase, which will be stored as the loop variable corresponding to the step that is currently processed. For example, starting with the statement *write all anagrams occurring in the list*, we identify *all anagrams* as a phrase indicating repetition, and `anagram` as the loop variable.

If an explicit indicator of repetition is not found, then we look for plural nouns as other potential indicators of repetition. Specifically, we seek plural nouns that are the head of their corresponding noun phrase. For instance, the statement *read the values* contains one plural noun (*values*) that is the head of its corresponding noun phrase, which is thus selected as an indicator of repetition, and it is also stored as the loop variable for this step. Note however that a statement such as *write the number of integers* will not be marked as repetitive, since the plural noun *integers* is not the head of a noun phrase, but a modifier.

In addition to the loop variable, we also seek an indication of how many times the loop should be repeated – if such information is available. This information is usually furnished as a number that modifies the loop variable, and we thus look for words labeled with a cardinal part-of-speech tag. For instance, in the example *generate 10000 random numbers*, we first identify *numbers* as an indicator of repetition (noun plural), and then find *10000* as the number of times this loop should be repeated. Both the loop variable and the loop count are stored together with the step information.

Finally, another important role of the loop finder component is the unification process, which seeks to combine several repetitive statements under a common loop structure, if they are linked by the same loop variable. For example, the actions [`generate` numbers] and [`count` numbers] will be both identified as repetitive statements with a common loop variable *number*, and thus they will be grouped together under the same loop structure.

4.3 Comment Identification

Although not playing a role in the execution of a program, comments are an important part of any computer program, as they provide detailed information on the various programming statements.

The comment identification step has the role of identifying those statements in the input natural language text that have a descriptive role, i.e. they provide additional specifications on the statements that will be executed by the program.

Starting with a step as identified in the step finding stage, we look for phrases that could indicate a descriptive role of the step. Specifically, we seek the following natural language constructs: (1) Sentences preceded by one of the expressions *for example*, *for instance*, *as an example*, which indicate that the sentence that follows provides an example of the expected behavior of the program. (2) Statements including a modal

verb in a conditional form, such as *should*, *would*, *might*, which are also indicators of expected behavior. (3) Statements with a verb in the passive voice, if this is the only verb in the statement[4]. (4) Finally, statements indicating assumptions, consisting of sentences that start with a verb like *assume*, *note*, etc. All the steps found to match one of these conditions are marked as comments, and thus no attempt will be made to turn them into programming statements.

An example of a step that will be turned into a comment is *For instance, 23 is an odd number*, which is a statement that has the role of illustrating the expected behavior of the program rather than asking for a specific action, and thus it is marked as a comment.

The output of the comment identification process is therefore a flag associated with each step, indicating whether the step can play the role of a comment. Note that although all steps, as identified by the step finding process, can play the role of informative comments in addition to the programming statements they generate, only those steps that are not explicitly marked as comments by the comment identification process can be turned into programming statements. In fact, the current system implementation will list all the steps in a comment section (see the sample output in Figure 2), but it will not attempt to turn any of the steps marked as "comments" into programming statements.

4.4 A Walk-Through Example

Consider again the example illustrated in figure 2. The generation of a computer program skeleton follows the three main steps highlighted earlier: step identification, comment identification, loop finder.

First, the step finder identifies the main steps that could be potentially turned into programming statements. Based on the heuristics described in section 4.1, the natural language text is broken down into the following steps: (1) [`generate` 10000 random numbers between 0 and 99 inclusive], (2) [`count` how many of times each number is generated], (3) [`write` these counts out to the screen], with the functions/parameters: `generateNumber(number)`, `count()`, and `writeCount(count)`.

Next, the comment finder does not identify any descriptive statements for this input text, and thus none of the steps found by the step finder are marked as comments. By default, all the steps are listed in the output program in a comment section.

Finally, the loop finder inspects the steps and tries to identify the presence of repetition. Here, we find a loop in the first step, with the loop variable `number` and loop count `10000`, a loop in the second step using the same loop variable `number`, and finally a loop in the third step with the loop variable `count`. Another operation performed by the loop finder component is unification, and in this case the first two steps are grouped under the same loop structure, since they have a common loop variable (`number`).

The output generated by the natural language programming system for the example in figure 2 is shown in figure 3.

[4] Note that although modal and passive verbs could also introduce candidate actions, since for now we target program skeletons and not fully-fledged programs that would compile and run, we believe that it is important to separate the main actions from the lower level details. We therefore ignore the "suggested" actions as introduced by modal or passive verbs, and explicitely mark them as comments.

```
#=============================================================
# Write a program to generate 10000 random numbers between 0 and
# 99 inclusive. You should count how many of times each number
# is generated and write these counts out to the screen.
#=============================================================

for($i = 0; $i < 10000; $i++) {
      # to generate 10000 random numbers between 0 and 99 inclusive
      &generateNumber(number)

      # You should count how many of times each number is generated
      &count()
}

foreach $count (@counts) {
      # write these counts out to the screen
      &writeCount(count)
}
```

Fig. 3. Sample output produced by the natural language programming system, for the example shown in figure 2

4.5 Evaluation and Results

One of the potential applications of such a natural language programming system is to assist those who begin learning how to program, by providing them with a skeleton of computer programs as required in programming assignments. Inspired by these applications, we collect a corpus of homework assignments as given in introductory programming classes, and attempt to automatically generate computer program skeletons for these programming assignments.

The corpus is collected using a Web crawler that searches the Web for pages containing the keywords *programming* and *examples*, and one of the keyphrases *write a program*, *write an applet*, *create a program*, *create an applet*. The result of the search process is a set of Web pages likely to include programming assignments. Next, in a post-processing phase, the Web pages are cleaned-up of HTML tags, and paragraphs containing the search keyphrases are selected as potential descriptions of programming problems. Finally, the resulting set is manually verified and any remaining noisy entries are thusly removed. The final set consists of 120 examples of programming assignments, with three examples illustrated in Table 1.

For the evaluation, we randomly selected a subset of 25 programming assignments from the set of Web-mined examples, and used them to create a gold standard testbed. For each of the 25 program descriptions, we manually labeled the main steps (which should result into programming statements), and the repetitive structures (which should result into loops). Next, from the automatically generated program skeletons, we identified all those steps and loops that were correct according to the gold standard, and

Table 1. Sample examples of programming assignments

Write a program that reads a string of keyboard characters and writes the characters in reverse order.
Write a program to read 10 lines of text and then writes the number of words contained in those lines.
Write a program that reads a sequence of integers terminated by any negative value. The program should then write the largest and smallest values that were entered.

correspondingly evaluate the precision and the recall of the system. Specifically, precision is defined as the number of correct programmatic structures (steps or loops) out of the total number of structures automatically identified; the precision for the step identification process was measured at 86.0%, and for the loop identification at 80.6%. The recall is defined as the number of correct programmatic structures from the total number of structures available in the gold standard; it was measured at 75.4% for the step identification component, and at 71.4% for the loop finder.

5 The Future: NLP for NLP

Natural language processing for natural language programming or natural language programming for natural language processing? We would argue that the benefits could go both ways.

Despite the useful "universal" aspect of programming languages, these languages are still understood only by *very few* people, unlike the natural languages which are understood by *all*. The ability to turn natural into programming languages will eventually decrease the gap between *very few* and *all*, and open the benefits of computer programming to a larger number of users. In this paper, we showed how current state-of-the-art techniques in natural language processing can allow us to devise a system for natural language programming that addresses both the descriptive and procedural programming paradigms. The output of the system consists of automatically generated program skeletons, which were shown to help non-expert programmers in their task of describing algorithms in a programmatic way. As it turns out, advances in natural language processing helped the task of natural language programming.

But we believe that natural language processing could also benefit from natural language programming. The process of deriving computer programs starting with a natural language text implies a plethora of sophisticated language processing tools – such as syntactic parsers, clause detectors, argument structure identifiers, semantic analyzers, methods for coreference resolution, and so forth – which can be effectively put at work and evaluated within the framework of natural language programming. We thus see natural language programming as a potential large scale end-user (or rather, end-computer) application of text processing tools, which puts forward challenges for the natural language processing community and could eventually trigger advances in this field.

References

1. BALLARD, B., AND BIERMAN, A. Programming in natural language: "NLC" as a prototype. In *Proceedings of the 1979 annual conference of ACM/CSC-ER* (1979).
2. BRILL, E. Transformation-based error driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics 21*, 4 (December 1995), 543–566.
3. DIJKSTRA, E. On the foolishness of "Natural Language Programming". In *Program Construction, International Summer School* (1979).
4. KATE, R., WONG, Y., GE, R., AND MOONEY, R. Learning to transform natural to formal languages. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)* (Pittsburgh, 2005).
5. LIEBERMAN, H., AND LIU, H. *Feasibility studies for programming in natural language*. Kluwer Academic Publishers, 2005.
6. LIU, H., AND LIEBERMAN, H. Metafor: Visualizing stories as code. In *ACM Conference on Intelligent User Interfaces (IUI-2005)* (San Diego, 2005).
7. LIU, H., AND LIEBERMAN, H. Programmatic semantics for natural language interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI-2005)* (Portland, OR, 2005).
8. PANE, J., RATANAMAHATANA, C., AND MYERS, B. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies 54*, 2 (2001).
9. SINGH, P. The Open Mind Common Sense project. *KurzweilAI.net* (January 2002). Available online from http://www.kurzweilai.net/.
10. TANG, L., AND MOONEY, R. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the 12th European Conference on Machine Learning (ECML-2001)* (Freiburg, Germany, 2001).