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

# 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](#_bookmark0). 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.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.

# 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](#_bookmark10)] 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. floatconstant(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](#_bookmark12)], 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](#_bookmark18)], the Web) and a suite of significantly advanced publicly available natural language processing tools.

For instance, Pane & Myers [[8](#_bookmark17)] conducted a series of studies with nonprogramming fifth grade users, and identified some of the programming models implied by the users natural language descriptions. In a similar vein, Lieberman & Liu [[5](#_bookmark14)] have conducted a feasibility study and showed how a partial understanding of a text, coupled with a dialogue with the user, can help nonexpert users make their intentions more precise when designing a computer program. Their study resulted in a system called METAFOR [[6](#_bookmark15)], [[7](#_bookmark16)], 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](#_bookmark13)], or previous versions of it as described in [[10](#_bookmark19)], 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](#_bookmark13)], or into queries that can be posed against a database of U.S. geography or job announcements [[10](#_bookmark19)].

# 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, objectoriented, and agentoriented the objectoriented and agentoriented formats most closely embody human storytelling intuition. Consider the task of programming a MUD[2](#_bookmark1) world by natural language description, and the sentence There is a bar with a bartender who makes drinks [[6](#_bookmark15)]. 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](#_bookmark16)]; 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](#_bookmark15)] natural language programming system that can render code following the descriptive paradigm, starting with a natural language text.

## Syntactic Correspondences

There are numerous syntactic correspondences between natural language and descriptive structures. Most of todays natural languages distinguish between various parts of speech that taggers such as Brills [[2]](#_bookmark11) 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 whodoeswhat structure, or subjectverbdirectObjectindirectObject (SVO) construction. Although the ordering of subject, verb, and objects differ across verbinitial (VSO and VOS, e.g. Tagalog), verbmedial (SVO, e.g. Thai and English), and verbfinal languages (SOV, e.g., Japanese), these basic three ingredients are rather invariant across languages, corresponding to an encoding of agentmethod and methodargument 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, nounmodifier relationships, named entities, can be also tackled using current stateoftheart natural language processing techniques, such as coreference tools, named entity annotators, and others.

Starting with an SVO structure, we can derive agentmethod and methodargument 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 superclass character for the more specific class Pacman.

2 A MUD (multiuser dungeon, dimension, or dialogue) is a multiplayer computer game that combines elements of roleplaying games, hack and slash style computer games, and social instant messaging chat rooms (definition from wikipedia.org).

## 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.

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](#_bookmark2), 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.

## SetBased Dynamic Reference

Setbased dynamic reference suggests that one way to interpret the rich descriptive semantics of compound noun phrases is to map them into mathematical sets and setbased 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 utterances 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. Setbased 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 setbased dynamic reference to discover implied repetition and loops.

Occams 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.

* + 1. There is a bar. (atom)
    2. The bar contains two customers. (unimorphic list of type Customer)
    3. It also has a waiter. (unimorphic list of type Person)
    4. It has some stools. (polymorphic list)
    5. The bar opens and closes. (class / agent)
    6. The bar is a kind of store. (agent with inheritance)
    7. Some bars close at 6pm, others at 7pm. (forks into two subclasses)

Applying the semantic patterns of syntactic correspondence, representational equivalence, setbased dynamic reference, and scoping description to the interpretation of natural language description, objectoriented code skeletons can be produced. These description skeletons then serve as a code model which procedures can be built out of. Mixedinitiative 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](#_bookmark15)] implementing the features highlighted in this section was evaluated in a user study, where 13 nonprogrammers and intermediate programmers estimated the usefulness of the system as a brainstorming tool. The nonprogrammers 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](#_bookmark14)] and [[8](#_bookmark17)] that natural language programming can be a useful tool, in particular for nonexpert 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.

# 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

Write a program to generate 1000 numbers between 0 and 99 inclusive. You should count how many times each number is generated and write these counts out to the screen.

@counts;

for($i = 0; $i < 10000; $i++) { &generateRandomNumber (\$number); &count($number);

}

$i = 0;

foreach $count (@counts) { &writeCount($i++, $count);

}

###### Subroutines ####### sub generateRandomNumber {

($ref) = @\_;

$$ref = 1 + rand(99);

}

sub count { ($number) = @\_;

$counts[$number]++;

}

sub writeCount { ($index, $count) = @\_;

print $index, , $count,\n;

}

Natural language (English) Programming language (Perl)

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](#_bookmark3), we would ideally like to generate a computer program as the one shown on the right side of the figure[3](#_bookmark4). 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 walkthrough example illustrating the process of converting natural language texts into computer program skeletons.

## 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](#_bookmark3)), 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 preprocessed, i.e. tokenized and partofspeech tagged using Brills tagger [[2](#_bookmark11)]. 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).

## 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](#_bookmark3)), implies that the resulting action statement of [generate random numbers] should be repeated 10000 times.