Natural Language Programming Using Class Sequential Rules

Cohan Sujay Carlos
Aiaioo Labs
Bangalore, India
cohan@aiaioo.com

Abstract

This paper presents a system for Natural Language Programming using Class Sequential Rules (CSR). The system recognizes a number of procedural primitives and operators. The domain of the system is presently limited to the world of numbers and operations on numbers. We evaluate the effectiveness of CSRs at the task of Natural Language Programming using an annotated corpus of programming instructions in natural language, achieving a precision and recall of 85% and 64% respectively. We also compare the performance of a system trained on annotated data with that of a system using handcrafted rules.

1 Introduction

Since the early days of computing, there have been those who have longed for a programming system wherein the use of a formal symbolism for programming was not strictly necessary. Though high level languages greatly improve the ease of programming, it might be easier for a user to be allowed to communicate with an application in a natural language rather than a formal language with a specialized syntax. Moreover, a significant fraction of the population of the world is not conversant in the English language or remains unfamiliar with the Roman alphabet, and for them, a system for programming in their native tongues would be of great benefit. It is also easy to foresee a future where speech recognition systems are so accurate and robust that users might seek to provide instructions by means of speech to their computers. Thus there seems to be a need for algorithms capable of accepting commands and programming instructions in unconstrained human languages.

Type	Arity	Example
if	2 or 3	If x is 2, say “Hi”
unless	2	Exit unless x is 2
while	2	While $x \leq 2 \ldots$
until	2	Till x is 2 add 1 to x
continuation	1	Also, increment y
assignment	2	Let x be 1
imperatives	0 to ∞	Display x
questions	1	What is y?
y/n questions	1	Is x equal to 2?

Table 1: Types of Procedural Primitives.

We present a Natural Language Programming system capable of accepting programming commands in a natural language, executing them and returning any requested results to the user. It is limited to the domain of real numbers and can be used to write programs to compute the values of various functions of real numbers, or to generate different number series. The system can recognize nine broad categories of procedural primitives, some of which are conditionals, loops, assignments and function calls (imperatives). The complete list of types of procedural primitives with their arity and examples of usage can be found in Table 1.

We also present and evaluate a novel approach for processing user instructions. Instead of the traditional programming language approach of using a parser to produce a parse tree starting from modules or blocks of programming instructions, we first break up the programming instructions into sentences. We then use a short text classifier to first classify each resulting sentence into one of the categories of primitives listed above. Then we perform entity extraction to obtain as many contiguous and non-overlapping word subsequences as the arity of the primitive recognized, and then repeat the process with each of the word subsequences. Our system is capable of learning to recognize programming instructions belonging to the categories described above from an annotated corpus. It can also use manually crafted rules for the recognition of commands from sentences in a natural language.

The output of the above process is a semantic parse of the program. The semantic parses of consecutive sentences are joined into blocks if continuations are indicated. For example, if the first line is "If x is 3, display x." and the second line is "Also, increment x.", the first line is semantically parsed into "x = 3 =>print(x)." and the second line is parsed to "also(++x)." Then the two lines are combined to get "x = 3 =>print(x) && ++x." The semantic parse is also an interlingua representation that can be stored, used to perform an automatic translation of the program from one language to another, or executed to obtain the output.

Mihalcea et al (2006) distinguish the two complementary programming tasks of *description* and *proceduralization*. The present paper deals with proceduralization, the process of constructing procedures out of steps, blocks, conditionals and loops. Of the procedural primitives listed in Table 1, the first two are *conditional* statement primitives; the next two are *loop* primitives; the fifth is used to construct *blocks* of statements by agglutination; the remainder are primitives for *steps*.

Imperatives tend to be function calls or commands that result in an action or change. Examples of imperatives include calls to "display the value of x" and "Go to the step marked 'Subroutine 1'."

Wh-questions and yes/no questions have the effect of displaying the value of a variable, literal or expression. Yes/no questions are distinguished from wh-questions because their argument is constrained to be a boolean expression whereas wh-questions can refer to non-boolean variables, literals and expressions.

The set of expressions supported by the system is listed in Table 2. It will be observed that many commonly used mathematical operations like exponentiation and logarithms are not included in the list. The list does however include the arithmetic and relational operators that have their own keywords in the C, C++ and Java programming languages. At present, only two of the logical operators, namely *or* and *and*, are supported.

The rest of this paper is organized as follows:

Expression	**Arity**
addition	2
subtraction	2
multiplication	2
division	2
modulus	2
divisible	2
equality	2
inequality	2
less than	2
less than or equal to	2
greater than	2
greater than or equal to	2
conjunction	2
disjunction	2
negation	1

Table 2: Expressions.

Section 2 presents related work on the topic of natural language programming. The annotated corpus used to evaluate the system and the annotation guidelines for the same are presented in Section 3. Section 4 describes the entity recognition algorithms in some detail. The entity recognition systems are evaluated in Section 5 on the annotated corpus, and the novel approach of training a natural language programming system from annotated text is compared with the approach of using manually crafted rules. The conclusions and future directions are presented in Section 6.

2 Related Work

Attempts to develop natural language programming systems have been made in the past, and a working prototype called "NLC" was described by Ballard and Biermann (1979). NLC was capable of accepting English commands in the imperative mood. It did not accept declaratives and interrogatives. Each input was required to begin with an imperative verb. However, NLC was capable of dealing with pronominal references, like the ones in the command "Put the average of the first four entries in that row into its last entry" and with procedure definitions and loops, though not with conditionals. A sample NLC program from Ballard and Biermann (1979) is provided in Table 3. An example of a looping instruction in NLC is the last line of the sample program that instructs the computer to repeat the preceding steps over other rows of the matrix under consideration. Biermann et

"Choose a row in the matrix"
"Divide its last entry by 3"
"Repeat for the other rows"

Table 3: Sample NLC Program.

Take the array [3, 5, 7, 4, 6, 2, 1].
Count from one up to the size of the array:
Go over the array from the beginning to the end minus the counter:
If the current element is bigger than the following element then exchange the current element with the following element.
Print "After: " and print the array.

Table 4: Sample Pegasus Program.

al (1983) also described attempts to provide commands to the NLC system using the Nippon Electric DP-200 Connected Speech Recognizer.

Knoell and Mezini (2006) described a programming language called Pegasus that lets users program using plain German or English statements. Pegasus uses a handcrafted phrase structure grammar to parse English or German language commands and convert them into an intermediate representation. The intermediate representation is turned into a Java program using a dictionary of code snippets or translated into a different language. Knoell and Mezini (2006) did not perform an evaluation of the system on a corpus of natural language programming commands. A sample program written in the Pegasus language (see Table 4) was provided in the paper to show what a Pegasus program to sort an array of numbers would look like.

Objections to Natural Language Programming have been recorded, most notably by Dijkstra (1978) in an article titled 'On the Foolishness of "Natural Language Programming"'. Knoell and Mezini (2006) on the other hand opine that a system capable of dealing with both, formal symbolism and natural text, might be better than one that only understands either.

A system for generating program skeletons from the text of programming assignments like the one shown in Table 5 was described by Mihalcea et al (2006). The system did not attempt to generate an executable program.

Pane and Myers (2000) studied how non-programmers would describe solutions to problems arising within a Pacman game, and from their study, Pane and Myers (2001) proposed several principles of usability for developing a programming system for children. Lieberman and Liu (2006) examine mixed-initiative dialog as a way of more precisely ascertaining user intention with respect to programming commands, again in the context of the Pacman game.

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.

Table 5: Programming Assignment.

The present work is also related to the area of natural language understanding. Shapiro (2001) describes a system that understands user statements about their beliefs about the world. This is in essense a form of descriptive programming, similar to that described in Lieberman and Liu (2005) who explore the possibility of using natural language descriptions as a representation for programs, and describe a system called "Metafor" capable of generating Python scaffolding (referred to as the visualization code) from them, though not fully specified programs.

There has been, to our knowledge, no prior attempt to measure the performance of a natural language programming system using a corpus of annotated programming statements in a natural language. There has also not been, to our knowledge, any prior attempt to use entity recognition on natural language commands to generate fully specified programs. We also believe that this is the first attempt to learn patterns of language for programming from an annotated corpus.

3 Data Set

At the time of writing, there was no corpus available for evaluating the performance of a system for procedural programming in a natural language. Therefore, a corpus was developed[1] consisting of sentences in the English language, which the contributors of the sentences perceived to be commands that ought to invoke the programming primitives listed in Table 1 or specify the expressions listed in Table 2.

[1]The annotated corpus can be downloaded from the URL http://www.aiaioo.com/corpora/vaklipi2011.

Order	Survey Question
1	How would you say "x = 2" in English?
2	How would you say "x != 2" in English?
4	How would you say "$x \leq 2$" ?
6	How would you say "$x \geq 2$" ?
9	How would you say "x multiplied by 2" ?
10	How would you say "x / 2" ?

Table 6: Survey Questions.

The sentences were collected by setting up an online survey on a website, sending out requests over social networks and using email. All the questions were presented to the user at the same time, and the instructions at the top said:

> Please type into the box under each of the following, one way of saying the same thing in English and click on the Submit link below it. You can repeatedly enter different phrases if you can think of many ways of saying the same thing. For example, to communicate the idea of $x = y$, you might say x and y are equal or assign the value y to x.

The survey consisted of a total of thirty questions, and took approximately thirty minutes to complete. All users were presented with the same set of questions and in the same order. The answers provided by previous users were not made available to subsequent survey takers.

A total of 3,517 sentences[2] was collected over a period of one month.

The annotation procedure involved recategorizing the submissions according to the procedural primitive or expression they most closely matched and marking the entity spans according to the following annotation rules:

- **Conditional primitives:** A complex sentence specifying a condition for performing an action is a conditional statement. It is classified as an *if conditional statement* if it is conditional upon a **positive** outcome of the conditioning expression and the action to be taken does not undeniably suggest repeatedly checking the conditioning expression as a precondition for performing the action. It is classified as an *unless conditional statement* if it is conditional upon a **negative** outcome of the conditioning expression.

- **Loop primitives:** A complex sentence specifying a condition that is repeatedly evaluated for performing an action until it is satisfied or denied is a loop statement. It is classified as a *while loop statement* if it is conditional upon a **positive** outcome of the conditioning expression. It is classified as an *until loop statement* if it is conditional upon a **negative** outcome of the conditioning expression.

- **Operations with side effects:** Requests to add a value to a variable or to subtract a value from a variable are considered increment and decrement operations if the value of the variable would normally be considered to have changed at the end of the operation. For example, "Add 2 to x" would be considered an increment operation, whereas "Add x to 2" would not. The latter would be considered an addition operation and not an increment operation since it does not suggest a change in the value of x whereas the former does.

- **Operations without side effects:** Sentences in the indicative mood, informative clauses and phrases are classified as operations that do not have side effects, if they are not expected to alter the value of a variable. For example, "x added to 5" and "x by 5" have no side effects.

- **Relational Operations:** The classification of relational operations like *less than* depends on the order in which the parameters are supplied. So, "x is less than 2" and "x is not greater than or equal to 2" are both valid ways of representing "$x < 2$", but not "2 is greater than x". The last example is classified as a *greater than* operator.

From the counts for the categories *if conditional statement* (95 before annotation and 118 after cleaup) and the *unless conditional statement* (64 before and 15 after) it appears to be the case that people strongly prefer using an *if conditional statement* to an *unless conditional statement*. Thus the number of test sentences in each of the categories is not balanced.

[2]The number 3,517 included blank sentences and names as well. When these were removed, we were left with about 3,100 sentences of which we annotated 3,000.

4 System Description

As the system is intended for use as a multilingual teaching tool for students of computer programming, especially for those who do not possess a knowledge of English or the Roman alphabet, it was thought desirable to use a method of processing natural language programming instructions that would permit the rules for processing instructions to be learnt from annotated text in a number of possibly very different human languages.

One approach that seemed very promising was Class Sequential Rules (CSR) as described by Hu and Liu (2006). CSRs have been shown to outperform existing methods at the task of associating opinions with product features (Liu et al, 2006). Moreover, CSRs can also be easily created by hand since they are a subset of cascading grammar rules such as those described by the Common Pattern Specification Language (CPSL) (Appelt, 1996), which incidentally was developed as a language for specifying finite-state grammars for the purpose of information extraction. It was intuitively felt that a less powerful formalism might not only suffice for the task of processing programming commands, but also prove learnable from an annotated corpus.

4.1 Class Sequential Rules

A Class Sequential Rule consists of a sequence of ordered tokens, that we indicate by the symbols $i_1 \ldots i_n$, for example, $I = < i_1 i_2 i_3 >$. A CSR matches a sentence only when each token in the CSR's token sequence matches a word token in the text under evaluation, in the right order. The CSR I will match a sentence s if and only if, in the sentence, there is a token s_3 that matches i_3, and this token follows a token s_2 that matches i_2. This second token must in turn follow a token s_1 that matches i_1. For example, I will match the sequence $< i_1 x_3 i_2 x_4 i_3 x_5 >$ but not $< i_2 i_3 >$ or $< i_3 i_2 i_1 >$. CSRs like I can be used for classification as follows: a number of mutually exclusive CSRs are assigned to each class and used in conjunction with a priority or ordering scheme to resolve conflicts when CSRs from different classes match the input.

CSRs can also have class labels $c_1 \ldots c_n$ in their sequences. A class label can match zero or more tokens in the sentence. The tokens that class labels match represent entites in the sentences when CSRs are used for entity extraction. For example, the CSR $J =< i_1 c_4 i_2 c_5 i_3 >$ will match the sentence $< i_1 x_3 x_4 i_2 i_3 >$. The two class labels in J, namely c_4 and c_5 will at the same time match the sub-sequence $< x_3 x_4 >$ and the empty sequence $<>$ respectively. As you can see, the actual matching is performed by the sequence tokens $i_1 \ldots i_n$. The class labels pick up the tokens in between.

If two class labels follow one another in quick succession (are not separated by a token in the sequence), for example $< i_1 c_2 c_3 i_4 >$, the extents of their spans are not well defined. The tokens that class labels match can be thought of as entities. Thus, in addition to classification, CSRs can be said to be capable of performing entity extraction.

The task of understanding a programming command given in natural language can be broken down into the two sub-tasks that CSRs perform: a) classifying the command into one of several categories of commands; and b) extracting the arguments for further processing. Both tasks can be performed by CSRs in a single step.

The algorithm for mining CSRs used in the present work is described in more detail in Hu and Liu (2006). Using the algorithm, it is possible to mine sequences with a given minimum *support*. For instance, given two sequences such as those shown in Table 7, and a minimum required support of 2, it is possible to extract all the patterns of length 1, 2 and 3 indicated in Table 8. CSRs are only a special case of sequential patterns. With CSRs, the sequential patterns mined are text tokens. Some algorithms for sequential pattern mining have been studied in Agrawal and Srikant (1995).

Sequences
$< c_2 keab >$
$< dc_2 kb >$

Table 7: Sequence database.

Length	Sequences
1	$< c_2 >$
2	$< c_2 k >$
3	$< c_2 kb >$

Table 8: Extracted sequences with support 2.

Other pattern exraction concepts closely related to CSRs include the surface patterns described by

Sequences
Also { x = 2 }.
Also, { x = 2 }.
Also { if x = 3 , ++x }.

Table 9: Three annotated sentences that demonstrate the inadequacy of CSRs for natural language programming.

Order	CSRs
1	Also , EXPRESSION .
2	Also EXPRESSION .

Table 10: Class Sequential Rules for the entity spans in Table 9.

Ravichandran and Hovy (2002), Hearst (1992), Snow et al (2005) and Lin and Pantel (2001).

In the course of developing a hand-crafted set of rules for natural language programming using CSRs, it was observed that the discriminative power of CSRs did not always suffice. For example, it was observed that CSRs could not accurately identify the entity spans in the three sentences listed in Table 9.

The reason is that the two rules in Table 10 are needed to match the spans in the first two sentences and these are ordered to fire one behind the other as shown in the table. Now, however, the first rule "Also , EXPRESSION ." incorrectly picks out the single entity span in the third sentence owing to the comma in the middle of the sentence, and there is no way to rectify the problem using a different number of CSRs or changing the ordering.

Thus, a family of rules with more discriminative power was needed. We attempted to extend the concept of CSRs to give them more discriminative power, as described in the next subsection.

4.2 Extended Class Sequential Rules

The extension that solved the problem described in the preceding section was that of allowing each token in the CSR to be an n-gram. The rules presented in Table 11 contain a special term "NONE" which indicates an n-gram constraint as follows:

Order	Extended CSRs
1	Also NONE , EXPRESSION .
2	Also EXPRESSION .

Table 11: Two CSRs of the extended variety.

- When the term "NONE" appears between two tokens, the tokens they match in a sentence must be consecutive tokens. This is equivalent to making tokens on either side of "NONE" part of an n-gram. In the example in Table 11, the first rule can only match sentences where a comma immediately follows the word "Also."

- The term "NONE" can also appear at the beginning of an extended CSR to indicate that the following token must appear at the beginning of the matched sentence.

- Similarly, it can appear at the end of an extended CSR to indicate that the preceding token can only appear as the last token of a matched sentence.

The set of rules in Table 11 will be seen to be able to match all three sentences' spans correctly.

4.3 Rule Selection and Ordering

The two parameters used to evaluate the suitability of a rule are *support* (the percentage the sentences belonging to a category that it correctly identifies as belonging to that category) and *confidence* (the percentage of matches of the rule that were right), which are analogous to precision and recall. Only those rules were selected whose support and confidence values on the training data both exceeded minimum thresholds. The selected rules were ordered as follows:

- Rules whose accuracy of span matching exceeded the threshold were placed first.

- Rules whose accuracy of span matching fell below the threshold followed.

- Within the two categories described above, longer rules went ahead of shorter rules.

4.4 Intermediate Representation

The intermediate representation is a programming language that mimics the ambiguities of the language used in mathematics. The intermediate representation differs from a conventional programming language like Python in the following ways:

- Terms which tend to be ambiguous in natural language remain so in the intermediate representation. For example, the 'assignment operator' doubles as the 'equal to operator'

Let x be 3. y is 9.
What is x times y?
While x is less than y, print x and then increment x.

Table 12: Sample commands in our system.

in many natural languages. This overloading poses no problem since the right form can be resolved from context.

- The intermediate representation also attempts to capture the logical operator priority of Indian languages. In many Indian languages, it is not possible to set *or*-phrases as subphrases of *and*-phrases. So, we have chosen a priority order for operations in the intermediate representation that naturally maintains the restriction.

A program written in the present system would look like that in Table 12.

5 Evaluation and Results

The natural language programming system was evaluated against the annotated corpus described in Section 3. The manual rules used in the evaluation were developed and fixed prior to the start of corpus collection to avoid the introduction of biases through any knowledge of the corpus to be tested on.

5.1 Categories

The categories that were used in testing were equality, inequality, less than, greater than, less than or equal to, greater than or equal to, addition, subtraction, multiplication, division, increment, decrement, if, while, unless, until, print, conjunctive, disjunctive, divisible and continuation.

The categories in the corpus that were left out of the evaluation were as follows:

- Three categories were left out because of the lack of manually crafted rules for those categories.
- Of the three categories recognized as the print command by the manual rules, only one was retained for the experiment.
- Two of the omitted categories were merely synomyms for boolean constants 'true' and 'false.'

Category	**Cnt**	**Precision**	**Recall**	**F1**
equality	298	79.0 ± 06	66.5 ± 19	71.9 ± 10
inequality	165	90.6 ± 14	78.6 ± 06	84.3 ± 09
less than	151	66.8 ± 10	88.4 ± 07	76.8 ± 08
$\leq$	137	99.1 ± 02	75 ± 13	86.0 ± 07
more than	158	76.6 ± 08	83.1 ± 06	79.6 ± 02
$\geq$	132	92.9 ± 05	80.8 ± 13	86.5 ± 09
addition	140	97.9 ± 04	61.2 ± 10	77.2 ± 06
subtract	113	92.5 ± 15	71.0 ± 06	80.8 ± 06
multiply	144	98.8 ± 02	64.1 ± 12	79.4 ± 08
division	143	89.8 ± 10	69.8 ± 08	79.2 ± 09
increment	136	92.5 ± 08	57.3 ± 08	72.8 ± 08
decrement	131	96.9 ± 06	23.5 ± 15	46.7 ± 15
if	118	84.2 ± 05	96.0 ± 08	89.8 ± 04
while	61	92.1 ± 02	88.0 ± 12	89.8 ± 11
unless	15	100 ± 00	60.7 ± 15	77.6 ± 09
until	86	98.8 ± 02	85.8 ± 15	91.9 ± 08
print	82	92.3 ± 06	33.9 ± 14	55.1 ± 09
and	68	52.8 ± 11	82.8 ± 14	66.1 ± 12
or	67	92.1 ± 08	37.8 ± 04	58.8 ± 03
divisible	66	92.7 ± 08	71.1 ± 18	80.7 ± 10
continue	48	78.3 ± 23	22.1 ± 11	40.0 ± 05

Table 13: Evaluation of CSR-EX.

- The modulus operator was left out of the evaluation because the manual rules treated the operator as 'modulus' whereas the question used in the survey used to develop the corpus had suggested that the operator was the 'absolute value' operator.

5.2 Experiments

The 3,000 sentences in the annotated corpus belong to 29 distinct categories of which 21 are used for evaluating the system. Support and confidence values of 0.0001 and 0.70[3] respectively were used during training (for rule discovery).

Since we performed 3-fold cross validation, three sets of experiments were conducted for each of the following settings, for a total of 9 experiments in all:

- Conventional CSRs (CSR-BL)
- Extended CSRs (CSR-EX)
- Manually crafted rules (CSR-Man)

[3]These values were manually chosen keeping in mind the small size of the corpus. The support threshold was chosen to be low enough to not affect rule selection. The confidence threshold was kept low enough to permit single failures (incorrect matches) from time to time.

Setting	Prec.	Recall	F1
CSR-Man	89.2 ± 3.7	64.8 ± 6.2	73.0 ± 4
CSR-BL	85.7 ± 4.5	65.3 ± 5.9	73.1 ± 4
CSR-EX	88.4 ± 3.4	66.5 ± 5.6	74.8 ± 3

Table 14: Categorization Evaluation.

Metric	CSR-Man	CSR-BL	CSR-EX
PSCS	52.4 ± 9.1	50.2 ± 8.4	49.7 ± 8.6

Table 15: Entity Span Matching Evaluation.

In all experiments, the Precision, Recall and F1 Score (the harmonic mean of Precision and Recall) were measured for each of the categories, as well as the overall accuracy of categorization. The Precision, Recall and F1 scores for the CSR-EX algorithm are presented in Table 13. In the second column of the table is listed the number of sentences used in the test. This value in some cases drops to as low as fifteen sentences. The confidence intervals are rather high, making it difficult to draw comparisons between algorithms based on this data. The average of these scores for all categories is reported in Table 14.

The accuracy of entity span boundary detection is measured as follows: A recall-based score for correct span detection is computed by dividing the number of sentences with perfectly identified spans by the number of sentences in the category. This score is reported as the PSCS (percentage of sentences with correct spans). This score is similar to but not quite the same as the PCS (percentage of correct scopes) metric used in Councill et al (2010).

The PSCS scores for the three algorithms are reported in Table 15. We observe from the results that for the corpus the evaluation was performed on, there is no significant difference between the algorithms evaluated.

The overall accuracy scores presented in Table 16 again reveal no significant differences between CSR-Man, CSR-BL and CSR-EX in their behaviour with respect to the data-set.

Metric	CSR-Man	CSR-BL	CSR-EX
Acc.	64.3 ± 7	64.4 ± 6	66.0 ± 4

Table 16: Accuracies.

6 Conclusions and Future Work

This paper presents a system for Natural Language Programming capable of recognizing a number of categories of procedural programming instructions in a natural language. The system uses Class Sequential Rules to convert a natural language representation of a program into an intermediate representation that can be executed. The system is capable of using manually crafted rules or rules learnt from an annotated corpus.

Since no corpus was available for evaluation of a system for Natural Language Programming, a corpus consisting of 3,000 sentences in twenty-nine categories (of which only twenty-one were used), was collected over the internet, cleaned, re-categorized, annotated with entity spans and made publicly available.

Since the existing formalism of Class Sequential Rules (CSR-BL) was not powerful enough to tease certain sets of sentences apart into the right categories, an extension to Class Sequential Rules was proposed (CSR-EX) and implemented.

Finally, the system was evaluated by three-fold cross validation using the corpus. Three settings of the system were tested: a) a setting where it used extended CSR-EX rules manually crafted before the collection of the corpus; b) a setting where it used CSR-BL rules learnt from the annotated corpus; and c) a setting where it used CSR-EX rules learnt from the annotated corpus. Precisions of around 85% and recalls of approximately 64% were measured with confidence intervals as large as 7%. The large confidence intervals make it impossible to establish if one of the approaches works better than the others with the present corpus and the present set of categories.

Future research could include an evaluation on a larger corpus, on more languages and on the system's ability to adapt to new domains. It would also be interesting to examine system accuracies with an increased number of categories covering more operations and functions. It might also be of interest to build a corpus of complete programs rather than individual sentences, to capture more variations in language.

Acknowledgments

We are very grateful to all the volunteers who contributed to the corpus and to Govind Sharma, Kartik Asooja, Dr. Pushpak Bhattacharyya and the reviewers for helpful suggestions and references.

References

Rakesh Agrawal and Ramakrishnan Srikant. 1995. *Mining Sequential Patterns*. In *Proceedings of the Eleventh International Conference on Data Engineering IEEE Computer Society Washington, DC, USA. 1995*.

Douglas E. Appelt. 1996. *The Common Pattern Specification Language*. In *Proceedings of a workshop on held at Baltimore, Maryland. 1996*, 23–30.

Bruce W. Ballard, Alan W. Biermann. 1979. *Programming in Natural Language: "NLC" as a prototype. Proceedings of the 1979 annual conference. 1979*, 228–237.

Alan W. Biermann, R. Rodman, Bruce W. Ballard, T. Betancourt, G. Bilbro, H. Deas, L. Fineman, P. Fink, K. Gilbert, D. Gregory, F. Heidlage. 1983. *Interactive natural language problem solving: a pragmatic approach*. In *ANLC '83, Proceedings of the first conference on Applied Natural Language Processing, 1983*.

Isaac G. Councill , Ryan McDonald , Leonid Velikovich. 2010. *Whats Great and Whats Not: Learning to Classify the Scope of Negation for Improved Sentiment Analysis*. In *Proceedings of the Workshop on Negation and Speculation in Natural Language Processing (NeSp-NLP 2010). 2010*.

Edsger W. Dijkstra. 1978. *On the foolishness of "Natural Language Programming"*. In *Program Construction. 1978*, 51–53.

Marti A. Hearst. 1992. *Automatic Acquisition of Hyponyms from Large Text Corpora*. In *Proceedings of the 14th conference on Computational Linguistics - Volume 2. 1992*.

Minqing Hu and Bing Liu. 2006. *Opinion Feature Extraction Using Class Sequential Rules. AAAI Spring Symposium on Computational Approaches to Analyzing Weblogs, Palo Alto, USA, March 2006*.

Roman Knoell and Mira Mezini. 2006. *Pegasus First Steps Toward a Naturalistic Programming Language*. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. 2006*, 542–559.

Henry Lieberman and Hugo Liu. 2005. *Metafor: Visualizing stories as code*. In *10th International Conference on Intelligent User Interfaces. 2005*.

Henry Lieberman and Hugo Liu. 2006. *Feasibility studies for programming in natural language. Lieberman, Paterno, Wulf (Eds.): End-User Development (Human-Computer Interaction Series Vol. 9), Springer, 2006*, 459–474.

Dekang Lin and Patrick Pantel. 2001. *Discovery of inference rules for question-answering*. In *Journal Natural Language Engineering archive Volume 7 Issue 4, Cambridge University Press New York, NY, USA. December 2001*.

Bing Liu, Minqing Hu and Junsheng Cheng. 2005. *Opinion Observer: Analyzing and Comparing Opinions on the Web*. In *Proceedings of the 14th International World Wide Web conference (WWW-2005), Chiba, Japan. 2005*.

Rada Mihalcea and Hugo Liu and Henry Lieberman. 2006. *NLP (Natural Language Processing) for NLP (Natural Language Programming)*. In *Proceedings of CICLing. 2006*, 319–330.

John F. Pane and Brad A. Myers. 2000. *The Influence of the Psychology of Programming on a Language Design: Project Status Report*. In *Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group, A. F. Blackwell and E. Bilotta, Eds. Corigliano Calabro, Italy: Edizioni Memoria, April 10-13 2000*, 193–205.

John F. Pane, Chotirat Ann Ratanamahatana and Brad A. Myers. 2001. *Studying the language and structure in non-programmers' solutions to programming problems*. In *International Journal of Human-Computer Studies Volume 54, Issue 2, February 2001*, 237–264.

Deepak Ravichandran and Eduard Hovy. 2002. *Learning Surface Text Patterns for a Question Answering System*. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 2002*.

Stuart C. Shapiro. 1989. *The CASSIE Projects: An Approach to Natural Language Competence*. *EPIA, 1989*, 362–380.

Rion Snow, Daniel Jurafsky, Andrew Y. Ng. 2005. *Learning Syntactic Patterns for Automatic Hypernym Discovery*. In *Advances in Neural Information Processing Systems 17. 2005*, 1297–1304.