

Programming With Unrestricted Natural Language

David Vadas and James R. Curran

School of Information Technologies

University of Sydney

NSW 2006, Australia

{dvadas1, james}@it.usyd.edu.au

Abstract

We argue it is better to program in a natural language such as English, instead of a programming language like Java. A natural language interface for programming should result in greater readability, as well as making possible a more intuitive way of writing code. In contrast to previous controlled language systems, we allow unrestricted syntax, using wide-coverage syntactic and semantic methods to extract information from the user's instructions.

We also look at how people actually give programming instructions in English, collecting and annotating a corpus of such statements. We identify differences between sentences in this corpus and in typical newspaper text, and the effect they have on how we process the natural language input. Finally, we demonstrate a prototype system, that is capable of translating some English instructions into executable code.

1 Introduction

Programming is hard. It requires a number of specialised skills and knowledge of the syntax of the particular programming language being used. Programmers need to know a number of different languages, that can vary in control structures, syntax, and standard libraries. In order to reduce these difficulties, we would like to express the steps of the algorithm we are writing in a more natural manner, without being forced into a particular syntax. Ideally, we want a *plain English description*.

We have built an initial prototype of such a system, taking unrestricted English as input, and outputting code in the Python programming language. There are many advantages of such a system. Firstly, any person that can write English, but not a programming language, would still be able to program. Also, it is often easier to write an English sentence describing what is to be done, than to figure out the equivalent code. Many programmers write in a pseudocode style that is almost English before elaborating on the details of an algorithm. There are

also many tasks that can easily be described using English sentences, but are much harder to express as code, such as negation and quantification.

Another advantage is that code written in English will be much easier to read and understand than in a traditional programming language. Quite often, it is a difficult task to read another programmer's code. Even understanding one's own code can be hard after a period of time. This is because without sufficient commenting — this is an explanation in plain English — one cannot tell what individual steps are meant to do together. In our system, the comments become the code.

Novice programmers could make great use out of such a system. They make simple syntax errors because they do not know the language well enough. Similarly, a novice programmer may know what function they want to use, but not its specific name and required arguments.

Finally, standard programming languages exhibit numerous technical details that are not evident in natural languages. Examples of this include typing, integer division and variable declarations. When we say in English $\frac{3}{5}$, we expect the result to be 0.6, not 0, as will result in many programming languages. These complications are a result of the computer's implementation, rather than the algorithm we are trying to describe. We would like to abstract away these issues, using information present in the English sentences to figure out the correct action to take.

2 An Example

We can see in Figure 1 two example programs that could be entered by a user. The code for the first program matches what is outputted by the current system, but the second is more complicated and does yet work correctly.

Looking at these examples, we can see a number of difficulties that make the problem hard, as well as form some intuitions that can help to solve the task. For example, the first line of both programs involves three function calls because of variable typ-

ENGLISH	PYTHON
read in a number add 2 to the number print out the number	<code>number = int(sys.stdin.readline().strip())</code> <code>number += 2</code> <code>print number</code>
read in 2 numbers add them together print out the result	<code>number1 = int(sys.stdin.readline().strip())</code> <code>number2 = int(sys.stdin.readline().strip())</code> <code>result = number1 + number2</code> <code>print result</code>

Figure 1: Some example English sentences and their Python translations.

ing. In Python, we must first read in a string, then strip away the newline character, and finally convert it to an integer. We can tell that integer conversion is required, firstly because of the name of the variable itself, and secondly, because a mathematical operation is applied to it later on. Of course, it is still ambiguous. The user may have expected the number to be a string, and to have the string 2 concatenated to what was read in. However, the code in Figure 1 is more likely to be correct, and if the user wants to use a string representation, then they could specify as much by saying: `read in a number as a string`.

Another problem to deal with is the referencing of variables. In the first program, it is fairly easy to know that `number` is the same variable in all three sentences, but this is not as easy in the second. For the first sentence of the second program, the system needs to interpret `2 numbers` correctly, and map it to multiple lines of code. Another complication is `them`, which references the previously mentioned variables. Finally, `result`, which does not appear in the second line, must still be part of the equivalent code, so that it can be used later.

One possibility that we could use to simplify the task that we are undertaking is to use a restricted natural language. However, we do not want to restrict the vocabulary available to a user, or force them to construct sentences in a specific way, as is the case for existing restricted natural languages (Fuchs and Schwitter, 1996). Of course, this means that we must then deal with the inherent ambiguity and the great breadth of unrestricted natural English. For this reason, we employ wide-coverage syntactic and semantic processing, that is able to process this extensive range of inputs. In order to resolve ambiguities, we can apply the intuitions we have described above. We may not be sure that the number should be treated as an integer, but this is more likely than treating it as a string. This is the conclusion that our system should come to as well.

3 Background

Clearly, the task we are undertaking is not trivial. Though there are a number of related systems to the one we propose, which have had success implementing a natural language interface for some task.

3.1 Natural Language Interfaces to Databases

The most popular task is a Natural Language Interface for a Database (NLIDB) (Androutsopoulos et al., 1995). This is because databases present a large amount of information, which both novice and expert users need to query. A specific query language such as SQL must be used, which requires one to understand the syntax for entering a query, and also the way to join the underlying tables to extract data that is needed. A NLIDB simplifies the task, by not requiring any knowledge of a specific query language, or of the underlying table structure of the database. We can see how this is similar to the English programming system that we are constructing. Both take a natural language as input, and map to some output that a computer can process.

There are a number of problems that exist with NLIDBs. Firstly, it is not easy to understand all the ambiguity of natural language, and as such, a NLIDB can simply respond with the wrong answers. As a result of this, many NLIDBs only accept a restricted subset of natural language. For example, in the NLIDB PRE (Epstein, 1985), relative clauses must come directly after the noun phrases they are attached to.

One feature of many NLIDBs, is the ability to engage the user in a dialogue, so that past events and previously mentioned objects can be referenced more easily. Two examples of this, anaphora and elliptical sentences, are shown in Figure 2.

Understanding that *it* refers to the ship, and that the female manager's degrees are again the subject of the question, reduces the amount of effort required by the user, and makes the discourse more natural. We also intend to maintain a discourse between the user and the computer for our own sys-

- ANAPHORA

> *Is there a ship whose destination is unknown?*
Yes.
> *What is it?*
What is [the ship whose
destination is unknown]?
Saratoga

- ELLIPTICAL SENTENCE

> *Does the highest paid female manager have
any degrees from Harvard?*
Yes, 1.
> *How about MIT?*
No, none.

Figure 2: An example of anaphora and an elliptical sentence

tem. This would also allow us to resolve much of the ambiguity involved in natural language by asking the user which possibility they actually meant.

3.2 Early Systems

One of the first natural language interfaces is SHRDLU (Winograd, 1972), which allows users to interact with a number of objects in what was called *Blocksworld*. This system is capable of discriminating between objects, fulfilling goals, and answering questions entered by the user. It also uses discourse in order to better interpret sentences from the user.

There were also a handful of systems that attempted to build a system similar to what we describe in this paper (Heidorn, 1976; Biermann et al., 1983). Most of these used a restricted syntax, or defined a specific domain over which they could be used. Our system should have much greater coverage, and be able to interpret most instructions from the user in some way.

More generally, we can look at a system that interprets natural language utterances about planetary bodies (Frost and Launchbury, 1989). This system processes queries about its knowledge base, but is restricted to sentences that are covered by its vocabulary and grammar. It deals with ambiguous questions by providing answers to each possible reading, even when those readings would be easily dismissed by humans. With our system, we will determine the most likely reading, and process the sentence accordingly.

3.3 Understanding Natural Language

One thing that we have not yet considered is how people would describe a task to be carried out, if they could use English to do so. The constructs and formalisms required by traditional programming languages do not apply when using a natural language. In fact, there are many differences between the way non-programmers describe a task, to the method that would be employed if one were using a typical programming language (Pane et al., 2001). Firstly, loops are hardly ever used explicitly, and instead, aggregate operations are applied to an entire list. These two methods for describing the same action are shown in Figure 3.

- AGGREGATE

sum up all the values in the list

- ITERATION

start the sum at 0

for each in value in the list

add this value to the sum

Figure 3: Finding the sum of the values in a list

Another point of difference comes in the way people use logical connectives such as AND and OR, which are not necessarily meant in the strictly logical way that is the case when using a programming language. There are also differences in the way that people describe conditions, remember the state of objects, and the way they reference those objects.

HANDS (Pane et al., 2002) is a programming language that has been designed with this information, and with the idea of providing a programming interface that is more natural to a human user. This system takes a controlled language as input, but still demonstrates a number of methods, such as the aggregate operations described above, which make it possible for people to describe the actions they want performed as if they were writing in English.

There are actually many ways in which natural language constructions map onto programming concepts. These *programmantic semantics* (Liu and Lieberman, 2004) can be seen in syntactic types, where nouns map to objects or classes, verbs map to methods, and adjectives to attributes of the classes. Using these concepts could allow us to more easily understand an English sentence, and map it to a corresponding code output.

Metafor (Liu and Lieberman, 2005) is a system

that uses these ideas, taking a natural language description as input. As output, the system provides *scaffolding code*, that is, the outline for classes and methods, and only a small amount of actual content. The code is not immediately executable, but can help the programmer in getting started.

NaturalJava (Price et al., 2000) is another natural language programming system that allows users to create and edit Java programs using English commands. Each sentence in the natural language input given to the system is mapped to one of 400 manually created case frames, which then extracts the triggering word and the arguments required for that frame. The frame can generate a change in the Abstract Syntax Tree (AST), an intermediate representation of the code, which is turned in Java code later.

This system has a number of problems that we intend to improve on. Firstly, it can only handle one action per sentence. Our prototype can detect multiple verbs in a sentence, and generate code for each of them. Also, the AST representation NaturalJava uses makes it hard to navigate around a large amount of code, since only simple movement operations are available.

Another problem with NaturalJava is that it maps to specific operations that are included in Java, rather than more general programming language concepts. This means that it is not adaptable to different programming languages. We intend to be more language-neutral. A user of our system should not need to look at the underlying code at all, just as a programmer writing in C does not need to look at the machine code.

4 Combinatory Categorical Grammar

Combinatory Categorical Grammar (CCG) is a type-driven, lexicalised theory of grammar (Steedman, 2000). Each word receives a syntactic category that defines its predicate-argument relationship with surrounding words. We can see a simple example of this in Figure 4.

Each word is assigned a category that defines how it is involved with other words in the sentence. These relationships are carried out through a number of rules, such as forward and backward application, which can be seen in the example. Additional rules such as composition and conjunction also allow the formalism to easily capture long-range dependencies. This is particularly important for our system, as the constructions used to describe programming instructions often contain non-standard constituents such as extraction, relativization, and coordination.

These possibilities result in a large number of

interpretations, as a single word can be assigned a different category depending on how it is used, and the words that surround it. However, the application of statistical parsing techniques for CCG have shown that it is capable of performing wide-coverage parsing at state-of-the-art levels (Clark and Curran, 2004).

5 English Code Corpus

In order to investigate the way that people would use English to describe a programming task, we elicited responses from programmers, asking them to describe how they would solve sample tasks. These tasks included finding the smallest number in a list, splitting a string on a character and finding all primes less than 100. The respondents were all experienced programmers, since computer science staff were all that were easily available. As a result of this, they tended to impose typical programming constructs on what they wanted to do, rather than using a simpler English sentence. For example, one respondent wrote *For each number in the list compare to min., when Compare each number in the list to the min. is more straightforward.* This demonstrates quite well the way that programming languages force us to use a specific unnatural syntax, rather than the freer style that a natural language allows. It also shows that experienced programmers can supply utterances that are less grammatically correct and therefore *harder* to process than what novices would be expected to write.

The corpus is comprised of 370 sentences, from 12 different respondents. They range in style quite significantly, with some using typically procedural constructs such as loops and ifs (complete with the non-sensical English statement: *end loop* in some cases), while others used a more declarative style.

We have semi-automatically tagged the entire corpus with CCG categories (called *supertags*). This process consisted of running the parser on the corpus, and then manually correcting each parse. Corrections were required in most sentences, as the way people express programming statements varies significantly from sentences found in newspaper text. An example of this is in Figure 5.

This sentence uses an imperative construction, beginning with a verb, which is quite different from declarative sentences found in newspaper text, and the earlier example in Figure 4. We can also notice that the final category for the sentence is $S[b] \setminus NP$, rather than simply S. Another difference is in the vocabulary used for programming tasks, compared to Wall Street Journal (WSJ) text. We find *if*, *loop*, and *variables* in the former, and

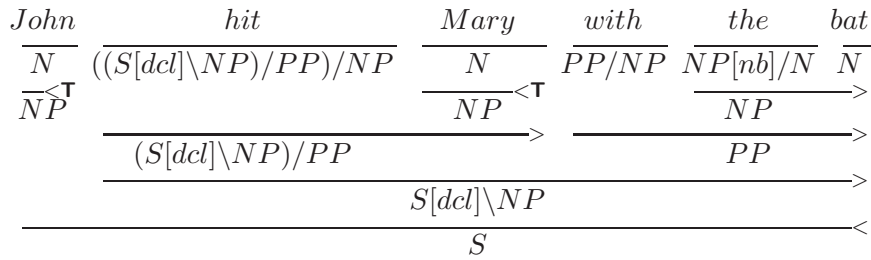


Figure 4: An example CCG derivation

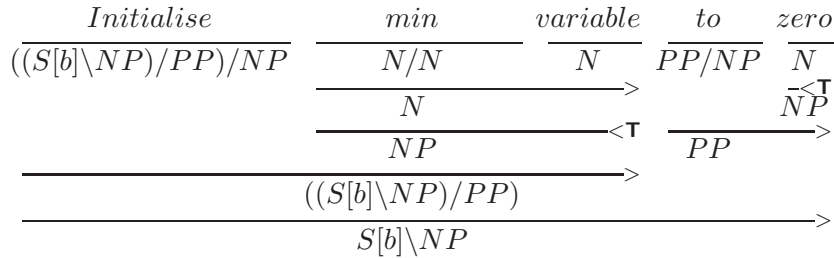


Figure 5: A CCG derivation for an English programming instruction

million, dollars, and executives in the latter. Particular words can also have different grammatical functions. For example: print is usually a noun in the WSJ, but mostly a verb while programming.

6 System Architecture

The system architecture and its components are shown in Figure 6.

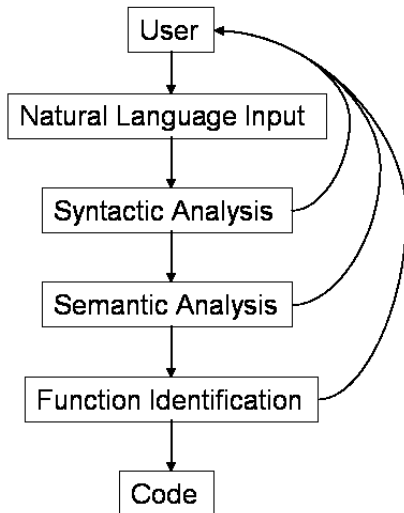


Figure 6: The system architecture

Firstly, the user will enter text that will be parsed by the CCG parser. We then translate the predicate-argument structure generated by the parser into a first-order logic representation of DRS predicates.

This gives us a more generic representation of the sentence, rather than the specific wording chosen by the user. The final step is to generate the code itself.

Throughout these three phases, we also intend to use a dialogue system that will interact with the user in order to resolve ambiguity in their input. For example, if the probability with which the parser gives its output is too low, we may ask the user to confirm the main verb or noun. This is especially important, as we do not intend for the system to be foolproof, but we do intend that the user should be able to solve the problems that they encounter, either through greater specification or rephrasing.

There are also a number of smaller tasks to be dealt with, such as anaphora resolution, and GUI construction. At this current stage though, we have only dealt with basic functionality.

We will now describe each of the components of the system in detail. Also, as we progress through each stage, we will follow the example previously shown in Figure 5. We will see how the processing we do manages to begin with this English input, and eventually output working Python code.

7 Parser

We use the C&C CCG parser (Clark and Curran, 2004) for this first stage of processing. This has the advantage of being a broad coverage, robust parser, that is able to extract long range dependencies reliably. We also have access to the code, and are thus able to make changes if needed, and are able to build new training models. In fact, we found that we

did indeed need to train a new model for the parser as a result of the differences between programming statements and typical newspaper text, as described above. If we look at our example sentence, we can see some of the problems quite well. Figure 7 shows the parse provided by the original model.

We can see that `Initialise` not been identified as a verb, but is instead tagged as a proper noun. `min` is also misclassified as a verb, when it is the noun. This highlights the fact that the parser does not expect the first word in a sentence to be a verb. We could not use this parse and expect to perform adequately in the following stages of the system.

For this reason we created and annotated the English code corpus, in order to provide training data and allow us to build a new, and better performing model. A similar process had been followed in Question Answering (QA) (Clark et al., 2004), because questions also show quite different syntactic properties to newspaper text. This technique produced a significant improvement for QA, and so we have reused this idea.

Following Clark et al., we used multiples of the English corpus, as it is quite small in comparison to the entire WSJ. These results are shown in Figure 8, for training with just the WSJ (original), with the WSJ and the English code corpus (1x code), and with the WSJ and multiples of the English corpus (5x, 10x, 20x). We show results for POS tagging and supertagging, on a word-by-word basis, and also the proportion of whole lines that are tagged correctly. We can see that as we add more copies of the English code corpus, all accuracies continue to improve.

These results come from both training and testing on the English corpus, and thus are not completely rigorous. However, it does demonstrate the data is fairly consistently annotated. As a fairer comparison, we conducted 10-fold cross validation on the 20x corpus, where each fold contained the 20 copies of one-tenth of the English code corpus, together with sentences from sections 2–21 of the WSJ. Each fold contained lines from throughout the English corpus and the WSJ. The results show that the accuracies from training with the English code corpus were still significantly greater than the original model. Finally, with this new model, our example sentence is parsed correctly, as shown in Figure 5.

8 Semantics

From the syntactic representation of the sentence, we wish to build a more semantically abstracted version of what the user wants to translate into code. The advantage of this is that we can more readily ex-

```
%%% Initialise 'min' variable to zero .
```

x4	x3	x5	x1	x2

thing(x4)				
'min'(x5)				
nn(x5,x3)				
variable(x3)				
initialise(x1)				
agent(x1,x4)				
patient(x1,x3)				
x2 >0				
to(x1,x2)				
event(x1)				

Figure 9: DRS for example sentence

tract the particular verbs and nouns that will become functions and their arguments respectively. Having a logical form also means we can apply inference tools, and thereby detect anomalies in the user’s descriptions, as well as including other sources of knowledge into the system.

The `csg2sem` system (Bos et al., 2004; Blackburn and Bos, 2005) performs this task, taking CCG parse trees as input, and outputting DRS logical predicates. A single unambiguous reading is always outputted for each sentence. The DRS for our example sentence is shown in Figure 9. We can see that the verb (x1) is identified by an event predicate, while the agent (x4) and patient (x3) are also found. One particular discriminating feature of the imperative sentences that we see, is that the agent has no representation in the sentence. We can also find the preposition (x2) attached to the verb, and this becomes an additional argument for the function.

This logical form also extracts conditions that would be found in if statements and loops very well. Figure 10 shows the DRSs for the sentence: `If num is -1, quit`. We can see the proposition DRS (the middle box) and the proposition itself (x2), which entails another verb (x3) to be interpreted. That is, we should carry out the verb `quit` (x1), if the proposition is true. Almost all if statements in the corpus are identified in this way.

9 Generation

Having extracted the functional verb and its arguments, we then need to find a mapping onto an equivalent line of code. The simplest technique, which the current system uses, consists of a list of primitives, each of which describes the specific verb in question as well as a number of arguments. If the semantic information matches perfectly with a

FUNCTIONAL VERB	ARGUMENTS	CODE TEMPLATE
read	input	<input> = int(sys.stdin.readline())
print	output	print <output>
add	addAmount, addTo	<addTo> += <addAmount>
initialise	variable, setting	<variable> = <setting>
set	variable, setting	<variable> = <setting>
assign	variable, setting	<variable> = <setting>
iterate	item, list	for <item> in <list>:

Figure 11: Primitives used for generation

of user-defined functions, just as a normal programming language would.

Looking back to our example sentence once more, we proceed to extract the predicate (*initialise*) and argument information (*min variable*, 0) from the DRS. This maps to the *initialise* primitive in Figure 11. The matching code, stored in the primitive, then comes out as:

```
min_variable = 0
```

This is clearly a suitable outcome, and we can say that for this case, the system has worked perfectly.

10 Future Work

There is a great deal of work still to be done, before we will have constructed a usable system. We intend to progress initially by expanding the generation component to be able to process most of the commands contained in the English code corpus. We also expect to do more work with the parser and semantic engine. For example, if we find that the coverage or accuracy of the parser is insufficient, then we can create more data for our corpus, or design specialised features to help disambiguate certain word types. Similarly, we may find that some information is not relevant or simply missing from the DRSs that are currently produced, in which case we would be required to extend the current system so that it can extract what is needed.

Once the three basic components described above function at a satisfactory level, then we will begin work on other components of the system. The largest of these is a dialogue component, which should solve a wide range of problems. These could include simple questions about the parse for a sentence:

```
> Blerg the number
Is Blerg a verb?
```

It could also help in resolving some ambiguity, or inquire about some missing information.

```
> Read in 2 numbers
```

```
> Add 2 to the number
Which number do you mean?
```

```
> Open a file for reading
What is the name of the file?
```

Anaphora resolution is another problem frequently encountered in the English code corpus we have collected. As discovered previously in the case of NLIDBs, having a system capable of dealing with this phenomenon makes it a great deal easier to use. For this reason, we intend to implement such a component for our final system.

Lastly, we intend to develop a GUI that allows a user to interact more easily with the system. Integrating the syntactic, semantic and generation components, together with a text editor, would allow the system to highlight certain functions and arguments. This would make it clearer to the user what the system is doing. The dialogue component in particular would gain a great deal from this, as it could be made clear what sentence or word was being clarified, as well as the context it was in.

11 Conclusion

Programming is a very complicated task, and any way in which it can be simplified will be of great benefit. The system we have outlined and prototyped aims to allow a user describe their instructions in a natural language. For this, a user may be asked to clarify or rephrase a number of points, but will not have to correct syntax errors as when using a normal programming language.

Using modern parsing techniques, and a better understanding of just how programmers would write English code, we have built a prototype that is capable of translating natural language input to working code. More complicated sentences that describe typical programming structures, such as if statements and loops, are also understood. Indeed, much of the work to be done involves increasing the coverage of the system in a general manner, so that it is able to understand a wider variety of user input. Once we have built a complete system that can make

some understanding of almost any input, we expect it to be usable by novice and experienced programmers alike.

Acknowledgements

We would like to thank members of the Language Technology Research Group and the anonymous reviewers for their helpful feedback. This work has been supported by the Australian Research Council under Discovery Project DP0453131.

References

- I. Androutopoulos, G.D. Ritchie, and P. Thanisch. 1995. Natural language interfaces to databases—an introduction. *Journal of Language Engineering*, 1(1):29–81.
- A. Biermann, B. Ballard, and A. Sigmon. 1983. An experimental study of natural language programming. *International Journal of Man-Machine Studies*, 18:71–87.
- P. Blackburn and J. Bos. 2005. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI Publications.
- J. Bos, S. Clark, M. Steedman, J.R. Curran, and J. Hockenmaier. 2004. Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING '04)*, Geneva, Switzerland.
- S. Clark and J.R. Curran. 2004. Parsing the WSJ using CCG and log-linear models. In *Proceedings of the 42nd Meeting of the ACL*, Barcelona, Spain.
- S. Clark, M. Steedman, and J.R. Curran. 2004. Object-extraction and question-parsing using CCG. In *Proceedings of the SIGDAT Conference on Empirical Methods in Natural Language Processing (EMNLP-04)*, pages 111–118, Barcelona, Spain.
- S.S. Epstein. 1985. Transportable natural language processing through simplicity – the PRE system. *ACM Transactions on Office Information Systems*, 3:107–120.
- R. Frost and J. Launchbury. 1989. Constructing natural language interpreters in a lazy functional language. *The Computer Journal. Special issue on lazy functional programming*, 32(2):108–121, April.
- N. E. Fuchs and R. Schwitter. 1996. Attempto controlled English (ACE). In *Proceedings of the First International Workshop on Controlled Language Applications*, pages 124–136.
- G. E. Heidorn. 1976. Automatic programming through natural language dialogue: A survey. *IBM Journal of Research and Development*, 20(4):302–313, July.
- H. Liu and H. Lieberman. 2004. Toward a programmatic semantics of natural language. In *Proceedings of VL/HCC'04: the 20th IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 281–282, Rome, September.
- H. Liu and H. Lieberman. 2005. Metaphor: Visualizing stories as code. In *Proceedings of the ACM International Conference on Intelligent User Interfaces*, pages 305–307, San Diego, CA, USA, January.
- J.F. Pane, C.A. Ratanamahatana, and B.A. Myers. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2):237–264, February.
- J.F. Pane, B.A. Myers, and L.B. Miller. 2002. Using hci techniques to design a more usable programming system. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC 2002)*, pages 198–206, Arlington, VA, September.
- D. Price, E. Riloff, J. Zachary, and B. Harvey. 2000. NaturalJava: A natural language interface for programming in Java. In *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, pages 207–211.
- M. Steedman. 2000. *The Syntactic Process*. The MIT Press, Cambridge, MA.
- T. Winograd. 1972. *Understanding Natural Language*. Academic Press.