NaturalJava: A Natural Language Interface for Programming in Java

David Price, Ellen Riloff, Joseph Zachary, Brandon Harvey
Department of Computer Science
University of Utah
50 Central Campus Drive, Room 3190
Salt Lake City, UT 84112 USA
+1 801 581 8224
{deprice,riloff,zachary,blharvey}@cs.utah.edu

ABSTRACT

NaturalJava is a prototype for an intelligent natural-language-based user interface for creating, modifying, and examining Java programs. The interface exploits three subsystems. The Sundance natural language processing system accepts English sentences as input and uses information extraction techniques to generate case frames representing program construction and editing directives. A knowledge-based case frame interpreter, PRISM, uses a decision tree to infer program modification operations from the case frames. A Java abstract syntax tree manager, TreeFace, provides the interface that PRISM uses to build and navigate the tree representation of an evolving Java program. In this paper, we describe the technical details of each component, explain the capabilities of the user interface, and present examples of NaturalJava in use.

Keywords

Intelligent user interfaces, information extraction, natural language processing, computer program editors, programming environments.

1. INTRODUCTION

Grappling with the syntax of a programming language can be frustrating for programmers because it distracts from the abstract task of creating a correct program. Visually impaired programmers have a difficult time with syntax because managing syntactic details and detecting syntactic errors are inherently visual tasks. As a result, a visually impaired programmer can spend a long time chasing down syntactic errors that a sighted programmer could have found instantly. Programmers suffering from repetitive stress injuries can have a difficult time entering and editing syntactically detailed programs from the keyboard. Novice programmers often struggle because they are forced to learn syntactic and general programming skills simultaneously. Even experienced programmers may be hampered by the need to learn the syntax of a new programming language.

We have created NaturalJava, a prototype for an intelligent, natural-language-based user interface that allows programmers to create, modify, and examine Java programs. With our interface, programmers describe programs using English sentences and the system automatically builds and manipulates a Java abstract syntax tree (AST) in response. When the user is finished, the AST is automatically converted into Java source code. The evolving Java program is also displayed in a separate window during the programming process so that the programmer can see the code as it is being generated.

The NaturalJava user interface has three components. The first component is Sundance, a natural language processing system that accepts English sentences as input and uses information extraction techniques to generate case frames representing programming concepts. The second component is PRISM, a knowledge-based case frame interpreter that uses a decision tree to infer high-level editing operations from the case frames. The third component is TreeFace, an AST manager that provides the interface used by the case frame interpreter to manage the syntax tree of the program being constructed.

Figure 1 illustrates the dependencies among the three modules and the user. PRISM presents a command line interface to the user, who enters an English sentence describing a program construction or editing directive. PRISM passes the sentence to Sundance, which returns a set of case frames that classify the key concepts of the sentence. PRISM analyzes the case frames and determines the appropriate program construction and editing operations, which it carries out by making calls to TreeFace. TreeFace maintains an internal AST representation of the evolving program. After each operation, TreeFace transforms the syntax tree into Java source code and makes it available to PRISM. PRISM displays this source code to the user, and saves it to a file when the session terminates. Figure 2 shows the user input and program display windows from a NaturalJava session.

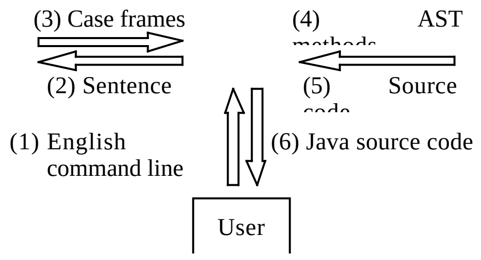
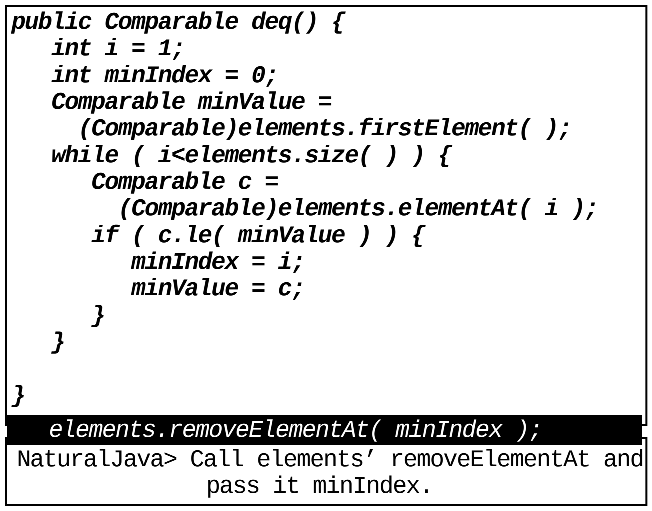


Figure 1. Architecture of NaturalJava



```
public Comparable deq() {
   int i = 1;
   int minIndex = 0;
   Comparable minValue =
     (Comparable)elements.firstElement( );
   while ( i<elements.size( ) ) {
      Comparable c =
        (Comparable)elements.elementAt( i );
      if ( c.le( minValue ) ) {
         minIndex = i;
         minValue = c;
      }
   }

}
   elements.removeElementAt( minIndex );
```

```
NaturalJava> Call elements' removeElementAt and
             pass it minIndex.
```

Figure 2. NaturalJava's program display and user input windows, created using first 11 steps of script from Figure 5.

2. NATURALJAVA USER INTERFACE

The goal of the NaturalJava interface is to allow programmers to write computer programs by expressing each command in natural language. For example, a user might ask the system to "create a for loop that iterates from 1 to 10." This form of interaction allows the programmer to give instructions without having to know the exact syntax required by the programming language.

2.1 Motivation and Background

Natural language (NL) interfaces can be plagued with two types of problems: natural language specifications can be ambiguous and incomplete, and natural language processing can be fragile because complete NL understanding is still beyond the state of the art. We addressed the first problem by limiting the role of inference in our system. NaturalJava recognizes NL commands that, while very similar to actual programming constructs, are expressed in English. This level of specification is relatively well defined, yet general enough that the programmer can focus on programming rather than syntax. The interface can detect when a command is incomplete (e.g., the terminating condition of a loop is missing) and prompt the user, but the role of inference in NaturalJava is mainly limited to the disambiguation of general verbs (e.g., "add" can refer to arithmetic or insertion).

We addressed the second problem of fragile natural language processing by using information extraction technology supported by a partial parser. Partial parsers are typically more robust and flexible than full parsers, which try to generate a complete parse tree for each sentence. Full parsers often fail on sentences that are ill-constructed or ungrammatical. Partial parsers are more robust because they do not have to generate a complete parse structure, but instead generate a flat syntactic representation of sentence fragments.

A few NL interfaces have been previously developed for programming (e.g., [1,7]). Perhaps the biggest difference between NaturalJava and previous systems is that NaturalJava allows users to generate and manipulate source code in a real programming language using an AST. Both MOON [7] and NLC [1] take immediate actions in response to natural language commands and do not maintain any internal representation of source code.

2.2 Understanding Commands Using IE

Information extraction (IE) is a form of natural language processing that involves extracting predefined types of information from natural language text. The goal is to identify information that is relevant to the task at hand while ignoring irrelevant information. Information extraction systems have been built for a variety of domains, including Latin American terrorism [4,5], joint ventures [5], microelectronics [5], job postings [2], rental ads [6], and seminar announcements [3].

For the NaturalJava interface, we used IE techniques to extract information related to Java programming constructs from the user's input. The natural language engine used by NaturalJava is a partial parser called Sundance, which was developed at the University of Utah. Sundance generates a flat syntactic representation of sentences and also can activate and instantiate pattern-based templates, or case frames. For the NaturalJava task, we manually designed 400 case frames to extract information about relevant programming constructs.

As an example, consider the sentence "Create a for loop that iterates from 1 to 10." Sundance begins by deriving a partial parse for this sentence, which involves part-of-speech disambiguation, syntactic bracketing, clause segmentation, and syntactic role assignment. Sundance then instantiates all active case frames to extract information from the sentence. The case frames represent local linguistic expressions revolving around verbs and nouns. Each case frame has a trigger word and an activating function that determines when it is applicable. For example, a case frame might be triggered by the word "iterates" when it appears as an active verb form. A case frame also has a type, which represents its general concept, and an arbitrary number of slots that extract information from local syntactic constituents.

Example 1 shows a case frame triggered by the verb "iterates." It contains four slots that extract information from the subject of the clause and from three prepositional phrases. For example, the subject of the clause will be extracted as the CONTROL_FLOW construct, while objects of the preposition "from" will be extracted as the start condition for the loop. The prepositional phrases may appear in any order, and any subset of these slots may be instantiated, depending on the input sentence.

```
iterates
(active_verb iterates)
type control_flow
{
   construct      SUBJECT
   loop_start     PREP (PREP=FROM)
   loop_end       PREP(PREP=TO)
   exit_condition PREP (PREP=WHILE)
}
```

Example 1. Example case frame template.

The final output of Sundance for the example sentence is shown in Example 2. Two case frames are generated, representing a CREATE concept and a CONTROL_FLOW concept. The CREATE case frame indicates that a for loop should be created, and the CONTROL_FLOW case frame specifies the control conditions for the loop. The CONTROL_FLOW case frame is instantiated from the template in Example 1. Notice that

Sundance did not extract an exit condition because there was no prepositional phrase for the preposition "while" in the sentence.

```
> Create a for loop that iterates from 1
to 10.

Caseframe CREATE_01(CREATE)
CREATE_TYPE:  "a FOR_LOOP"

Caseframe ITERATES_01(CONTROL_FLOW)
CONSTRUCT:    "a FOR_LOOP"
LOOP_START:   "&&1"
LOOP_END:     "&&10"
```

Example 2. Case frames generated by Sundance.

2.3 Mapping Case Frames into Instructions

The Programming Instruction Synthesis Module (PRISM) provides NaturalJava's command line user interface. The user enters commands as sentences or sentence fragments. Commands can add new information to the abstract syntax tree that represents the evolving Java program, delete information from the AST, modify information in the AST, navigate through the AST, or request information about the contents of the AST. PRISM preprocesses the input by replacing special symbols, such as math tokens, with appropriate words, and then passes the resulting sentence to Sundance for information extraction. Sundance instantiates and returns a set of case frames as explained earlier.

Sundance generates 27 types of case frames; three representative types are summarized in Figure 3. The type of a case frame indicates the nature of the user's request or the type of information found within the case frame's extracted strings. For example, case frames of type CREATE are triggered by verbs such as "create" and "declare." If these words occur as the primary verb, they indicate the need to create a method, class, or variable. Similarly, case frames of type NAVIGATION are triggered by verbs such as "move" and "go," and indicate the need to move the editing focus within the AST.

Type:	**Example triggers:**	**Example sentences:**
create	create	Create a class.
	declare	I would like to declare a method.
	want parameter	I want a parameter.
math	plus	x plus y.
	subtract	Subtract a from b.
	increment	Increment count.
multi purpose	add	Add a parameter. Add 3 to x.
	make	Make a class called C. Make C public.

Figure 3. Example case frame types.

PRISM divides the of the case frame processing into two tasks: determining the type of action the user desires, and retrieving the necessary information from the case frames to carry out that request. Two assumptions simplify the task of determining the action to be taken. First, PRISM assumes that each request by the user contains only one type of action. Second, PRISM assumes that the first verb in the request provides the information necessary to determine the type of action desired by the user. For example, "assign x plus y to z" is a valid request, but "add x to y and assign it to z" will not be processed correctly.

PRISM uses a decision tree to convert the case frames extracted by Sundance into actions to be taken on the AST. The first level in this decision tree sorts the case frames into action types¾such as declarations and requests for information¾based on the type of the primary case frame. PRISM deals with verbs that can be used in more than one type of command, such as "make" and "give," with an action disambiguation method. This method examines information in the extracted strings to determine the proper action to take. For example, PRISM determines that "make a double called my_double" is a variable declaration but that "make my_name public" changes a property of a data member. If the primary case frame does not contain the necessary information, then PRISM discards it and examines subsequent case frames. For example, given "make x equal to y," PRISM discards the "make" case frame and examines the next case frame for "equal," which suggests that the command is an assignment.

Subsequent levels of the decision tree examine the primary case frame's trigger word and extracted strings to further subdivide the command. PRISM often uses the current editing context of the AST to further constrain the nature of the user's request.

2.4 Creating and Manipulating ASTs

TreeFace is a Java class that is used by PRISM to create and manipulate objects that encapsulate AST representations of Java source files. TreeFace provides constructors that create empty ASTs and that initialize ASTs by parsing Java source files. TreeFace also provides methods that navigate through, add content to, perform generic editing operations on, and return information about an AST. In response to instantiated case frames produced by Sundance, PRISM composes appropriate sequences of TreeFace constructor and method invocations.

A TreeFace object also keeps track of the current editing context. PRISM uses this context to determine where in an AST a particular editing operation should take effect. The user must often change the editing context, much as the user of a standard editor must often change the current selection. Since the editing context is always some subtree of an entire AST, changes to the editing context are expressed in terms of motion through a tree. TreeFace's navigation methods include methods to push into and pop out of the body of a compound construct, and methods to move to the siblings of the constituents of a compound construct.

TreeFace provides content creation methods that create new classes and interfaces, member variables, methods, local variables, compound statements such as loops and conditionals, and simple statements such as assignments and returns. It also provides methods that allow the user to change certain attributes of existing constructs. For example, the user can make a member private.

TreeFace's generic editing operations allow the user to delete the current selection and to undo recent modifications to the AST. TreeFace also provides operations that report the state of the AST. These operations allow the user to request

1. Create a public method called deq that returns a Comparable.

2. Declare an int called i and initialize it to 1.
3. Declare an int called minIndex and initialize it to 0.

4. Declare a Comparable called minValue and initialize it to elements' firstElement cast to a Comparable.
5. Create a loop that iterates while i is less than elements' size.
6. Declare a Comparable called c and initialize it to elements' elementAt applied to i cast to Comparable.
7. Create an if statement controlled by c's le applied to minValue.
8. Assign i to minIndex.
9. Assign c to minValue.
10. Leave this loop.
11. Invoke elements' removeElementAt with minIndex as a parameter.
12. Return minValue.

Figure 4. Excerpt from first user's script

1. I would like to define a public method that is named deq and that returns a Comparable.
2. Declare an int variable named i that is initialized to 1.
3. Declare an integer variable named minIndex that has an initial value of 0.
4. Add a Comparable variable named minValue which is equal to elements' firstElement but that is cast to a Comparable.
5. Declare a loop and have it iterate while i < elements' size.
6. Add a Comparable named c, initialize it to elements' elementAt, pass in i, and cast to a Comparable.
7. If c's le when passed minvalue.
8. minIndex gets i.
9. minValue gets c.
10. Exit the loop.
11. Call elements' removeElementAt and pass it minIndex.

12. Please return minValue.

Figure 5. Excerpt from second user's script

information about the AST, such as the list of variables currently in scope. PRISM uses this capability to answer questions posed by the user.

3. USER INTERFACE EXPERIMENTS

The prototype interface is fully implemented and can be used to produce Java code. During a programming session, the system displays one window that accepts program editing commands and another that displays the Java source code as it is being generated. One of our main goals was to allow flexibility in natural language input, so two of the authors used NaturalJava to write exactly the same program. The first user defined a priority queue class, and the second user tried to generate exactly the same source code while using different natural language sentences. Excerpts from the transcripts of the user sessions are shown in Figures 4 and 5, and the Java code that resulted is shown in Figure 2.

4. LIMITATIONS AND FUTURE WORK

There are a number of limitations that we hope to address in future research. Two that will be relatively easy to rectify are to generalize PRISM to eliminate the two assumptions described in Section 2.3 and to add more case frames to increase the vocabulary of Sundance.

NaturalJava supports a large but incomplete subset of Java. It does not support array declarations, for example, because we have not yet added the required case frames and associated logic to Sundance and PRISM. Similarly, it does not support nested classes because we have not yet built the required AST support into TreeFace. Such limitations are a result of our depth-first development strategy, and will be addressed in future versions.

We plan to do more extensive experiments with NaturalJava to get experience with a wider variety of users. Our preliminary experiments, for example, have highlighted the need for compiler and debugger feedback to be coordinated with the AST interface.

The current implementation of NaturalJava is best suited for writing new source code and doing local, statement-level editing. Expression-level editing, direct navigation to distant sections of source code, and global program modifications are unsupported. For example, the only way to modify an expression is to delete and replace the statement that contains it. Moving the editing focus to a distant source code location can require a long sequence of AST traversal operations. Renaming a variable requires editing its declaration as well as every occurrence of it. The major thrust of our future research will center on addressing these issues.

We believe that our approach is sufficiently general that our interface could be easily modified to support other programming languages. We hope to demonstrate this once NaturalJava is more fully developed. The most useful future development would be to base the user interface on *spoken*, as opposed to *written*, natural language. This is, of course, a significant research challenge.

5. ACKNOWLEDGMENTS

We wish to thank David Bean and Jeff Lorenzen for help with Sundance and JNI. This research was supported under NSF grants IRI-9509820 and IRI-9704240.

6. REFERENCES

[1] Biermann, A., Ballard, B., and Sigmon, A. An Experimental Study of Natural Language Programming. *International Journal of Man-Machine Studies*, Vol. 18, pp. 71-87, 1983.

[2] Califf, M. E. Relational Learning Techniques for Natural Language Information Extraction. Ph.D. Dissertation, Tech. Rept. AI98-276, Artificial Intelligence Laboratory, The University of Texas at Austin, 1998.

[3] Freitag, D. Multistrategy Learning for Information Extraction, In *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998.

[4] Riloff, E. Automatically Generating Extraction Patterns from Untagged Text. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[5] Riloff, E. An Empirical Study of Automated Dictionary Construction for Information Extraction in Three Domains. *Artificial Intelligence* 85:101--134. 1996.

[6] Soderland, S. Learning Information Extraction Rules for Semi-structured and Free Text. To appear in *Machine Learning*, 1999.

[7] Wonisch, M. Ein objektorientierter interaktiver Interpreter fur naturalichsprachliche Programmierung. Diploma Thesis. Lehrstuhl fur MeBtechnik, RWTH Aachen, June 1995.