English for Spoken Programming

Benjamin M. Gordon
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico 87131
Email: bmgordon@cs.unm.edu

George F. Luger
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico 87131
Email: luger@cs.unm.edu

***Abstract*—Existing commercial and open source speech recognition engines do not come with prebuilt models that lend themselves to natural input of programming languages. Prior approaches to this problem have largely concentrated on developing spoken syntax for existing programming languages. In this paper, we instead describe a new programming language and environment that is being developed to use "closer to English" syntax. In addition to providing a more intuitive spoken syntax for users, this allows existing speech recognizers to achieve improved accuracy using their prebuilt English models. Our basic recognizer is built from a standard context-free grammar together with the CMU Sphinx pre-trained English models. To improve its accuracy, we modify the language model during runtime by factoring in additional context derived from the program text, such as variable scoping and type inference. While still a work in progress, we anticipate that this will yield measurable improvements in speed and accuracy of spoken program dictation.**

I. Introduction

The dominant paradigm for programming a computer today is text entry via keyboard and mouse. Keyboard-based entry has served us well for decades, but it is not ideal in all situations. People may have many reasons to wish for usable alternative input methods, ranging from disabilities or injuries to naturalness of input. For example, a person with a wrist or hand injury may find herself entirely unable to type, but with no impairment to her thinking abilities or desire to program. What a frustrating combination!

Furthermore, with the recent surge in keyboardless tablet computing, it will not be long before people want to program directly on their tablets. Today's generation of tablets are severely limited in comparison to a desktop system, suitable for viewing many forms of content, but not for creating new text-based content. Newly announced products already claim support for high-resolution screens, multicore processors, and large memory capacities, but they still will not include a keyboard. It is certainly possible to pair a tablet with an external keyboard if a large amount of text entry is needed, but carrying around a separate keyboard seems to defeat the main ideas of a tablet computer.

What is really needed in these and other similar situations is a new input mechanism that permits us to dispose of the keyboard entirely. Humans have been speaking and drawing for far longer than they have been typing, so employing one of these mechanisms seems to make the most sense. Products such as Apple's Siri have demonstrated the usefulness of systems built around non-keyboard inputs, but they do not incorporate facilities that are of use to programmers of traditional computer languages. In this paper, we consider the problem of enabling programming via spoken input by describing a new programming language and associated programming environment that are intended to model English more closely.

This paper is organized as follows: Section II describes the problem of spoken programming and related research. Section III provides a brief overview of our proposed language and some sample programs. Finally, section IV describes the current implementation status and future work.

II. Background

Once we have accepted the arguments from section I that programming via speech is a desirable thing to be able to do, the next problem becomes how to achieve this. A plausible first attempt would be to open up a traditional IDE

and attempt to use an existing dictation package to begin writing a program. Unfortunately, existing commercial and open source speech recognition engines do not come with prebuilt models that lend themselves to natural input of programming languages. For example, to input this potential line of code:

```
X := 5;
```

one common commercial package requires the following sentence:

```
X space bar colon equal sign
five semicolon new line
```

Just imagine dictating a full line containing all the punctuation and non-English words found in a real programming language! Clearly, this will not do for anything more than the most trivial programs.

Researchers have approached this problem from two opposite directions. In the first approach, researchers have studied how programmers speak existing languages and attempted to create models allowing dictation of traditional languages. Even though the computer only works on textual input, programmers often have to "speak code" to each other (e.g. pair programming). Thus, spoken forms of programming languages do exist in some sense, and these can be used for dictation. This approach was demonstrated by Begel and Graham in Spoken Java [1], [2]. When using Spoken Java, the user interacts with a modified Eclipse environment in which they can create Java code through a Java-like spoken syntax. The spoken syntax was created by studying how several expert programmers read Java code aloud.

A similar approach was taken by Désilets, Fox and Norton in VoiceCode [3], [4]. Instead of being a direct spoken interpretation of a programming language, VoiceCode recognizes a higher-level metalanguage and translates the recognized code patterns into native code output using templates. It can produce C++ and Python natively, and additional languages can be supported by adding appropriate grammars and code templates. A demo Java implementation was created by Masuoka [5]. In addition, it uses simple context sensitivity to determine when the user is speaking a programming construct versus an identifier. For example, the word "if" is interpreted as part of an if-then construct only when it is spoken with the cursor on a blank line.

On the completely opposite end of the scale, researchers have attempted to accept unstructured English as input and convert it to code output. This approach is exemplified by Price in NaturalJava [6]. In NaturalJava, the user describes his code using natural English speech. The system interprets the user's speech to produce a working Java program that is intended to follow his description. While the system was never fully implemented, Price did perform a semester-long "Wizard of Oz" study and found that approximately 80% of the phrases used by beginner programmers were stock phrases that could be easily interpreted without attempting the full generality of understanding English [7].

III. A New Programming Language

Rather than attempting to create a syntax or mechanism for dictating programs using an existing language, in this paper we describe a new programming language and environment that is being developed to use "closer to English" syntax. The language is not English; it is a simple, imperative-style language similar to C that is described by a typical grammar and compiled with a traditional compiler. However, the language constructs are described with English words and phrases that sound "natural" based on an informal survey of other department members. In particular, punctuation and other non-pronounceable items have largely been removed or replaced with English wording.

This organization provides two immediate benefits. First, the mapping between the words the user needs to speak and the text that appears on the screen is immediate and unambiguous. The authors of both VoiceCode and Spoken Java found that users naturally have multiple ways of describing the same symbols [1], [3], and both had to build in allowances for this. The user has less to learn and less to remember by eliminating these ambiguities. Moreover, our grammar and recognizer rules can be simplified.

Second, the use of English-like phrasing as the language syntax allows existing speech recognizers to achieve improved accuracy using their prebuilt English models. Our basic recognizer

is built from a standard context-free grammar together with the CMU Sphinx [8] pre-trained English models. With no additional training or modification to the recognizer, we have been able to dictate small–but complete–programs. To improve the accuracy of the system and scale to larger programs, we modify the language model during runtime by incorporating additional context derived from the program text, such as variable scoping and type inference. Further details are provided in section IV.

Our language is an imperative, compiled language with static typing. We use type inference to avoid requiring the user to speak type annotations. Because the focus of our research is spoken programming rather than language design, we have implemented a limited—but Turing-complete—set of constructs that are sufficient to write complete programs. We have omitted many advanced features that would be expected in a real-world language. It supports the following constructs:

1) String literals and variables
2) Integer literals and variables
3) Variable assignment
4) Simple arithmetic expressions and string operations
5) Function definitions
6) Function calls, including recursion
7) Indefinite loops (*while-do*/*repeat-until* equivalent)
8) Conditional statements (*if-then-else* equivalent)
9) Simple I/O (*printf* and *scanf* equivalents)

A. Examples

We provide a few examples here to illustrate programming in this language.

1) Trivial: Like programs written in C, each program's starting routine is called *main*. It takes no arguments and returns an integer exit status to the operating system. This is the shortest possible program:

```
define function main
taking no arguments
as
  return 0
end function
```

Note that indentation and spacing are provided to aid readability. They are not required for compilation.

2) Hello, World: Here is the quintessential first program:

```
define function main
taking no arguments as
  print the string "Hello, World"
  return 0
end function
```

3) Variables: We can read input from the user and perform simple calculations. Note that we do not have to specify the types of X or Y. They are inferred from their use in an integer expression.

```
define function main
taking no arguments as
  read X
  set Y to X * 2
  print Y
  return 0
end function
```

4) Conditionals and Looping: We provide typical *if-then* statements and indefinite (*while*) loops. This program prints the odd integers from 1 to 10 in a strangely convoluted way. Notice that every *if-then* must have a matching *else*, so we have a statement that does nothing. This program could have been written with less code if the language had definite loops. This is an example of a common feature that we have omitted to keep the focus on spoken programming instead of the language itself.

```
define function main taking
no arguments as
  set i to 1
  set odd to 1
  while i <= 10 do
    if odd = 1 then
      print i
      new line
    else
      nothing
    end if
    set odd to 1 - odd
    set i to i + 1
  end while
end function
```

5) Fibonacci: As a final example, we put everything from above together to make a program that does something slightly more useful. This program prints the first n Fibonacci numbers, one per line (where n is entered by the user):

```
define function main
taking no arguments as
  read limit
  set i to 1
  while i < limit do
    print the result of
      calling fib with i
    new line
    set i to i + 1
  end while
  return 0
end function

define function fib
taking arguments n as
  if n < 3 then
    return 1
  else
    set f to the result of
      calling fib with n - 1
    set g to the result of
      calling fib with n - 2
    return f + g
  end if
end function
```

IV. Speech Models

We implemented a programming environment for spoken programming as an Eclipse [9] plugin using CMU Sphinx [8] as the speech recognition engine and ANTLR [10] to implement the compiler. Other than the addition of spoken programming input, we have tried to leave the standard Eclipse experience in place. The user is able to make use of all the normal processes for compiling and running programs, managing resources through the explorer view, etc. In addition, she can dictate spoken programs. We have not implemented a voice-driven editing interface; instead, we currently rely on the built-in Eclipse editors and require the user to use the keyboard for post-dictation edits.

Because the language syntax is composed nearly entirely of English words, we have been able to dictate complete programs without any further modifications to the speech models that come with Sphinx. However, the accuracy might best be described as less-than-stellar. We have found that with some practice, we can dictate programs with an error only every couple of lines, but this is still frustratingly frequent.

The key to improving accuracy in English recognition research has been the incorporation of different types of context, either through sheer volume of training data or through domain- and task-specific knowledge. For example, Apple's Siri is able to handle many queries in natural English by interpreting them in the context of tasks that the user is likely to be performing on a phone. This same principle should apply to speech in the context of programming; we are implementing the two specific cases of variable scoping and type inference.

When a user speaks the name of a symbol or identifier, the principle of locality suggests that they are more likely to be speaking the name of something nearby than something defined in a far-off scope. We have implemented this idea by keeping a separate language model on a per-scope basis. The primary language model is based on the grammar describing the language and does not change. Ideally, it would only be necessary to adjust the probabilities of identifiers as they come in and out of scope. However, Sphinx does not allow modification of models while they are in use. It does allow the use of multiple models at runtime, so we use this facility to dynamically switch between tweaked models whenever the scope changes.

Similarly, knowledge of the type system allows us to improve the recognition of identifiers that are spoken as part of expressions and function calls. For example, if the user says "two times bar," there may be several identifiers that sound like *bar* in scope. However, we know that the other argument of *times* must be an integer to match the first argument. Thus, we can restrict our list of potential identifiers to those that are type-compatible with integers. The same idea can be used to constrain arguments passed in function calls.

The type of dynamic language model described for scoping above does not work for typing, be-

cause we would be potentially changing the model after nearly every word when the user speaks an expression. Instead, we apply the typing context as a second, higher-level model. Normally, we retrieve the unique best result from Sphinx whenever it has recognized part of a valid programming construct. When we detect that the user is speaking an expression or function argument, we instead ask it for a list of candidate hypotheses. We can then type-check the results and eliminate those hypotheses that include invalid combinations of identifiers.

V. Status and Future Work

We have implemented a full compiler and the base spoken programming environment for our language. With some practice and careful enunciation, it is possible to dictate simple programs at least as quickly as typing. The initial work of designing the addition of our scope- and type-based context into Sphinx is complete, but the implementation is still in progress. We anticipate this being ready for wider use by the end of this year.

We are also in the process of designing a user study to quantitatively measure the effectiveness of our added context. While the implementation work is not yet complete, our preliminary results based on using the system ourselves lead us to believe that the additional context we are incorporating will produce measurable improvements in accuracy and speed of spoken program input.

An important topic for future work is the integration of spoken navigation and editing commands. People do not typically write an entire program from the top down, so they need a way to move around in their source code. Currently both editing and navigation must be done with the keyboard or mouse, which we have found to be the clumsiest part of using our system.

References

[1] A. Begel and S. L. Graham, "Spoken programs," *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pp. 99 – 106, 2005.

[2] ——, "An assessment of a speech-based programming environment," *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pp. 116–120, 2006.

[3] A. Désilets, D. Fox, and S. Norton, "Voicecode: an innovative speech interface for programming-by-voice," *CHI'06 extended abstracts on Human factors in computing systems*, pp. 239–242, 2006.

[4] ——, "Voicecode programming by voice toolbox," http://sourceforge.net/projects/voicecode/, Retrieved January 3, 2011.

[5] C. Masuoka, "Java programming using voice input: Adding java support to voicecode," http://www.cs.umd.edu/Honors/reports/cmasuoka_HonorsProjectReport.pdf, Fall 2008.

[6] D. Price, E. Rilofff, J. Zachary, and B. Harvey, "NaturalJava: a natural language interface for programming in Java," in *Proceedings of the 5th international conference on Intelligent user interfaces*, ser. IUI '00. New York, NY, USA: ACM, 2000, pp. 207–211. [Online]. Available: http://doi.acm.org/10.1145/325737.325845

[7] D. E. Price, D. Dahlstrom, B. Newton, and J. L. Zachary, "Off To See The Wizard: Using A "Wizard Of Oz" Study To Learn How To Design A Spoken Language Interface For Programming," in *In Proceedings of the Frontiers in Education Conference*, 2002, pp. 2–23.

[8] "CMU sphinx - speech recognition toolkit," http://cmusphinx.sourceforge.net/, Retrieved January 15, 2011.

[9] "Eclipse: The eclipse foundation open source community website," http://www.eclipse.org/, Retrieved January 10, 2011.

[10] "ANTLR Parser Generator," http://www.antlr.org/, Retrieved April 13, 2012.