Building a Spoken Programming Language: Exploiting Program Structure to Disambiguate Utterances

Benjamin M. Gordon[a], George F. Luger[a,*]

[a] *Department of Computer Science, University of New Mexico, Albuquerque, New Mexico*

Abstract

Existing speech recognition systems perform poorly for the task of dictating program text (spoken programming). The systems that have been built tend to require extensive user training or exhibit high error rates. We have developed a new programming language with "English-like" syntax that produces major improvements in accuracy. In this paper, we examine how providing the speech recognition engine with additional context in the form of scoping and type inference further improves users' ability to dictate programs. Finally, we provide empirical validation of our system through a small study of programmers using the system to dictate programs.

Keywords: spoken programming, programming-by-voice

1. Introduction

Speech recognition systems have improved dramatically in the last decade for many real-world uses, but one important area that has not received much attention is spoken programming, i.e., the task of entering and editing a program's source code via dictation. Since programming in most mainstream languages is a heavily text-oriented endeavor, our initial intuition might suggest that we could reuse the same systems that work so well for word processing and email to dictate program source code. Unfortunately, this is not the case, or at least not in terms of practical daily usability.

A preliminary question for this type of research is why somebody would want to program through spoken input. Why not use the perfectly good keyboard already attached to their computer? We believe that the answer is twofold. First, consider all the people who have an injury or disability that impairs their ability to type. A person in this situation may have as much mental capacity and intelligence as anybody, but still be unable to write a program without a second person to do their typing. This is both inefficient and disempowering.

*Corresponding author

Email addresses: `bmgordon@cs.unm.edu` (Benjamin M. Gordon), `luger@cs.unm.edu` (George F. Luger)

For the second half of the answer, we suggest that with the rise of mobile computing, this group of people–those who know how to program but can't type very well–has expanded to include everybody using a smartphone or tablet. No matter how proficient a typist a person might be, they aren't likely be very quick at entering large quantities of text on a tablet's soft keyboard, especially when that text includes all the non-alphanumeric characters used in a typical programming language.

A number of other researchers have developed systems that facilitate spoken programming in existing languages, such as Java. These systems have enjoyed some success, but they also face many difficulties arising from the challenge of bridging human language and the artificial, mathematical syntaxes of computer languages. We have taken an alternative approach of designing a new programming language with syntax that more closely matches English sentence patterns. The change in language syntax alone shows promising results in terms of increased recognition accuracy [1]. In this paper, we describe how further improvements have been achieved by exploiting the program structure embedded in the source code to create additional context for the recognizer.

In section 2, we provide a description of the spoken programming task and related research in the area. We also briefly describe our system. In section 3, we give further details of our chosen syntax and the implementation of the system. In section 4, we describe the reasoning behind adding type inference and the mechanism for doing so. In section 5, we describe our testing methodology and study design. In section 6, we describe the results of running our study. Finally, in section 7, we provide some conclusions and ideas for future work.

2. Background

We first give a brief history of spoken programming and other related research. In this section, we also describe the key factors in the design of our new system, in particular the language syntax and its integration into a programming environment.

2.1. Spoken Programming

The idea of spoken programming is not new. Researchers have been experimenting with at least partial voice control of a programming environment since the Formulate [2] language in the 1980s.

In more recent years, there have been several notable attempts to build spoken programming environments. In 2006, Désilets, Fox, and Norton created VoiceCode [3, 4], a combination of the Emacs text editor with a commercial speech recognizer that supports speech-based input of multiple existing programming languages. VoiceCode handles the differences between language syntaxes by defining a generic higher-level spoken syntax to describe program features like classes and functions. When the user speaks one of these items, VoiceCode uses a language-specific template to generate appropriate code for the language being produced.

Another spoken programming environment called Spoken Java was created by Begel and Graham around the same time [5]. Their environment, being focused on Java only, used a spoken syntax that was meant to evoke the types of phrases that developers were observed to use when describing Java code to each other. The Spoken Java environment filled in all the necessary punctuation and syntax that was missing from users' utterances. They evaluated their environment with a small user study and found that it worked overall, but with approximately a 50% word error rate (WER) [6].

Arnold, Mark and Goldthwaite proposed a system called VocalProgramming [7]. Their system was intended to take a context free grammar (CFG) for a programming language and automatically create a "syntax-directed editor," meaning an editor that makes use of the programming language syntax for navigation and editing rather than merely allowing text entry. For example, they suggested that a navigation command like "move down to the next statement" would move to the next statement in the program regardless of how many lines that represented. After the initial paper, the system appears to have never been implemented, nor have any follow-on papers been published. It was described as abandoned by Begel and Graham [5].

The final project we will mention here is a fascinating presentation by Tavis Rudd at PyCon 2013 [8]. After Rudd developed severe RSI, he created a system in which he could rapidly create and edit LISP programs using various grunts and nonsense syllables to control indentation, function calls, and so on. His system required him to spend several months learning an entirely made-up language in order to use it productively, but his demo during his presentation makes a convincing case for the idea that special-purpose spoken syntax can be very effective when it matches the programming language.

2.2. An Alternative Approach

The previously described projects have each enjoyed some degree of success. However, they all suffer from either large user training periods or low input accuracy. We believe this is largely due to the major differences between the typical programming language syntax and the spoken phrases that humans use to "speak" them. Existing speech recognition programs are trained on and heavily biased towards producing valid English sentences, so they perform poorly when asked to recognize phrases that do not correspond to proper sentences.

One potential approach to this problem would be to develop a large corpus of spoken programming fragments, then retrain a recognizer on this data. Because such data does not currently exist, this would be labor-intensive, as well as losing much of the decades of research and tuning that has gone into existing English recognizers. Our approach has been to instead adapt the programming language syntax to more closely match typical English sentence patterns. For example, instead of using "x = 5;" we would say "set x to 5". Instead of defining a function with "void f(a,b,c)" we would say "define function f taking parameters a and b and c". More complete examples can be found throughout this paper, and the complete language grammar is available elsewhere [1].

3. New Language Implementation

In this section, we describe the details of the new programming language syntax. We also provide a brief description of the implementation.

3.1. A Brief Tour of the Language

As mentioned, our language syntax is intended to evoke the feeling of speaking English. While not being fully general English sentences, the individual statements are built from phrases that are either grammatical—though restricted—English phrases or are at least plausible chunks that would be spoken by an actual person. The specific syntax of each construct was determined in a fairly ad hoc manner by informal surveys and discussions around the UNM Computer Science department. Most constructs had a single dominant popular choice of phrasing. In the cases where there were alternatives with no clear winner, we chose one based on what felt most natural to ourselves. While certainly not a rigorously formal syntax development process, we feel that the end result is natural enough to be easily learnable and comfortably speakable by a fluent English speaker. The truth of this claim will be explored in Section 6.

The most popular languages today (Java, C, Python, etc.) are all imperative languages. The functional and logic programming paradigms offer some appealing benefits for the design of a language, but the use of these would require many potential users to learn both a new spoken syntax and a new programming paradigm at once. This would have made it difficult to design a study that tests the effectiveness of the syntax unless we limited testers to people who already had existing functional programming experience. We felt that this restriction would have unduly restricted our ability to run the study described in section 5.3. Therefore, to maximize accessibility of the language, we chose to make it an imperative language.

Even in the imperative paradigm, there is a wide variety of capabilities provided by the programming language, ranging from assembly to modern high-level object-oriented languages. Because the purpose of this research is not programming language features per se, we have chosen to omit many of the more complex features. The language we have built provides similar functionality to C in terms of control structures and data types, but we have incorporated some higher-level features such as automatic memory management and type inference. The individual features will be described below.

Also like C, our language is statically typed. While the common trend among recent languages seems to be in the direction of dynamic typing (consider Javascript, Python, etc.), the use of dynamic typing discards a significant amount of contextual information that can be used prior to runtime. Because the use of extra context to improve programming performance is one of the key ideas in this research, we did not wish to sacrifice this source of information; thus, static typing.

However, just because the language is statically typed does not mean we must force the programmer to litter their code with type declarations. Instead, we use type inference [9] to automatically recover this information at compile time (or even at program dictation time). This allows the programmer to focus on their algorithms rather than on the specifics of variable types, while still allowing us to make use of type information for speech recognition purposes.

While the combination of features above do not make an interpreter impossible, they do mean that the entire source file must be processed to infer types and resolve forward references before execution can begin. Because the design of runtime language environments is also outside the scope of this research, we found it more expedient to simply generate an executable after the initial processing rather than create an additional runtime interpreter.

The overall combination of the decisions above results in a minimalist, C-like language for spoken programming. Though minimal, our language is Turing complete [1]. Nevertheless, it is not meant to be a "production-ready" language in the sense of being usable to build large, real-world systems. Many features that are considered indispensable for modern software engineering are missing, but this is not a problem for our initial purposes of investigating spoken programming.

Next, we describe the individual language features and their syntax. The following gives an intuitive description of the syntax and the reasons for choosing it. A full formal grammar of the complete language is available elsewhere [1].

3.1.1. Variables and Data

The core of any imperative language is state manipulation, and the standard way to provide this is through variables and variable assignments. In order to assign variables, it is also necessary to have a syntax for literals of each of the supported types. It seems necessary to support string and integer variables in order to perform any non-trivial tasks. Booleans can be trivially emulated using integers, so they are not included as a separate type. Similarly, characters can be treated as a special case of strings. We also provide arrays of the above, but more sophisticated combinations such as sum types (variants), product types (records/structures), and user-defined types are all omitted.

The basic statement to create and update variables is called `set`. It takes an expressions and stores it into the named variable or array reference. If the variable did not previously exist, this creates the symbol and assigns its initial type. If the variable already existed, the type of the new value must be compatible with the previous type.
Examples:

```
set n to 1
set y to the string Hello
set element n of x to y
```

Note that string literals are introduced by the words "`the string`." The string value extends from there to the end of the line. When speaking, this is indicated by pausing at the end of the string. This is one of the few concessions to the use of non-visible features to indicate meaning. This decision was necessitated by practicality; further study is needed to determin a more complete mechanism for delimiting spoken strings.

In order to perform computations with its state, the language needs to include statements that can combine and manipulate variables. For numeric variables, this consists of arithmetic expressions. We have implemented addition, subtraction, multiplication, and division. More complicated expressions can be built using standard order of operations to enforce precedence.

Sensible basic manipulations for string variables are concatenation, substring extraction, and simple search (like the *strstr* function in C). Other string functionality can be easily built on top of these basic operations, so additional functionality has not been built in.

Modern languages also typically provide automatic memory management. In languages with pointers and references, this is often provided through garbage collection or reference counting. In this language, there are no pointers and no references, so memory for scalars can be automatically allocated and deallocated based on the call stack at runtime. Arrays with dynamic length can require dynamic memory allocation at any time, but this is handled automatically by a small runtime support library we have written. Since the array variables cannot outlive their scope, they do not introduce any additional difficulties with deallocation. Thus, our language does not need to provide any explicit memory management facilities.
Examples:

```
set x to 5
set z to x + 1
set element z of y to z + 5 * x
```

3.1.2. Functions

Since Dijkstra's famous letter on structured programming [10], it has been considered inconceivable to program in any language that did not include structured programming facilities. The most important such constructs are functions and loops.

Our language supports definition of simple functions that accept zero or more positional parameters with the types described above. Functions return single values, again of any of the supported variable types. Functions may also return nothing (the equivalent of *void* in C). More advanced parameter schemes like optional parameters, default values, and named parameters are not present. Because the language does not include pointers, all function parameters are passed and returned by value.
Examples:

```
define function main taking no arguments as
    ...
end function

define function foo taking arguments x as
    return x + 1
end function
```

```

    define function bar taking arguments m and n as
        ...
    end function

    define function baz taking arguments m and n and z as
        ...
    end function
```

There are two small oddities to notice here: the word `arguments` is always plural, and argument lists are joined with `and` instead of commas. This is intended both to keep the syntax regular and to avoid non-spoken content. While "`taking arguments x`" may be ungrammatical, we have found that it causes no problems for users; additionally, the speech recognizer can automatically correct if the person says "`taking argument x`," so there is no impediment to users' potential word choice here. The unusual list structure is slightly more disconcerting for people, but the alternatives were to explicitly say "comma" between words (which would be at cross-purposes to our design goals) or to use pauses to distinguish between identifiers (which was neither robust nor easy for users to discover).

Unlike variable symbols, which are created at their time of first assignment, all functions are conceptually in scope from the first line of code. Naturally, this requires the compiler to make multiple passes over its internal abstract syntax tree (AST), but this is not uncommon. More importantly, it enables forward references to functions (but not variables) that have not yet been defined at the point of their reference, and it enables recursion. Functions may call themselves recursively, and they may be mutually recursive through some chain of intermediate functions.

Like variables, function parameter types and return types are determined via type inference. However, functions are not polymorphic; all of a particular function's return types must resolve to a single type solution. This limitation is not inherent in the language grammar, but is simply an implementation limitation caused by the underlying code that the compiler generates. This limitation could be lifted without modifying the language syntax, but because polymorphism itself is not part of this research, we have not done so.

In addition to defining functions, a programmer must be able to call them. In order to keep the natural English feel of the language, we found it necessary to provide two syntaxes for this. For functions where the result is thrown away (called for purposes of side effects), the syntax is `call` *`function`*. If the function takes arguments, one must append `with` *`arguments`*. For function calls where the result will be stored in a variable or used in an expression, the syntax is `the result of calling` *`function`*. Arguments are appended the same way as the `call` statement.
Examples:

```
    call f
    call f with 1
```

```
call f with 2 and x
set x to the result of calling f
set x to 5 + the result of calling factorial with 10
```

From a language design standpoint, it seems mathematically inelegant to offer two syntaxes for the same purpose (of calling functions). However, the alternative requires either verbal gymnastics or the creation of unnatural sounding statements such as “set x to 5 + call f”. Does that statement mean “set x to 5, plus (also) call f”, or does it mean “call f, add 5 to the result, and store that value in x?” We know from the grammar restrictions that it means the second, but our casual daily use of “plus” to mean “also” can lead to the wrong intuition. Rather than either of these alternatives, we provide the two separate function call syntaxes.

3.1.3. Looping

There are two basic kinds of loops: definite and indefinite. In a definite loop, the loop body will execute some fixed number of times, while an indefinite loop is repeated an arbitrary number of times as long as its guard condition remains true. It is simple to simulate a definite loop using an indefinite loop plus a counter variable, but it is impossible to simulate an indefinite loop using a definite loop. Therefore, the language includes syntax for indefinite loops and omits definite loops.

The syntax for indefinite loops is the very typical `while` ***`condition`*** `do`. Exactly as one would expect, the condition is evaluated each time the `while` statement is reached, and the body will be executed repeatedly as long as it remains true.
Examples:

```
while 1 = 1 do
    ...
end while

while x < 5 do
    ...
end while
```

The body of the loop is a nested scope. It inherits the symbols visible in the parent scope, but any new variables created within the loop body will not be visible beyond the execution of the loop.

3.1.4. Conditionals

In theory, it is possible to simulate conditional statements (*if-then-else*) using a combination of indefinite loops and extra guard variables. The same reasoning that justifies omitting definite loops could also justify omitting conditionals. However, conditionals are so common, and the emulation is so cumbersome in comparison, that this seems unreasonable. Therefore, in the interests of avoiding the Turing tar pit [11], we have also included a built-in syntax for conditional statements. Like many functional languages, our `if-then` always includes an `else` and an explicit terminating `end if`. The primary purpose of this decision is simply consistency, although it does also have the benefit of avoiding the "dangling-else" problem.

Because this may then result in unwanted `else` blocks, we also provide the statement `nothing`, which allows the user to state that nothing should happen. This statement is primarily intended for use in otherwise-empty `else` blocks, but it is valid anywhere a statement can go.
Examples:

```
if x < 5 then
    ...
else
    ...
end if

if x > 1 and x < 10 then
    ...
else
    nothing
end if

if not z = y or y >= 0 then
    nothing
else
    ...
end if
```

Like the body of the `while` loop, both the true and false branches of the conditional introduce new nested scopes.

3.1.5. Booleans

Although we declared in section 3.1.1 that Boolean variables were unnecessary due to the inclusion of integers, both conditionals and indefinite loops require Boolean tests. Thus, simple Boolean tests and logic must be provided. These values can be used in `while` conditions and `if` statements, but cannot be directly stored into variables or otherwise treated as first-class values.

For all variables, we include tests for equality via the normal equals sign (=). For numeric variables, we also permit numeric comparisons (<, >, <=, >=). In order to complete Boolean logic, we will also need to be able to combine these with `AND`, `OR`, and `NOT`.

Examples can be found in the previous two sections.

3.1.6. Input and Output

Any program that cannot perform input and output is only useful for turning your CPU into a space heater. Thus, we have naturally provided some basic I/O facilities. The `print` statement evaluates an expression and prints the result to

standard output. It does not produce a new line after its output. We provide the `new line` statement to explicitly control line breaks.
Examples:

```
print 5
print x * 2
print element 5 of a
new line
print the string Hello
```

The `read` statement reads a value from the user and stores it in the named variable. Like the `set` statement, this automatically creates the variable if it did not previously exist within the current scope. The type of the variable is automatically determined by the type inference procedure based on other uses of the variable, and the `read` statement automatically coerces the input into the correct type.
Examples:

```
read x
read element 1 of y
```

All modern operating systems provide support for redirection of a program's standard input and output to and from files. Therefore, we have chosen not to provide explicit support for file I/O. Network sockets, shared memory, and other interprocess communication facilities are also omitted.

Most languages support some kind of external library linkage. This is a vital feature that would need to be present in any programming language made for serious use, but it is not necessary to demonstrate the viability of spoken programming. In addition, interfacing to external libraries written in other languages re-introduces the problem of trying to create a spoken syntax for other programming languages. Therefore, this feature has been omitted, but this will be an important area for future study.

3.2. Programming Environment Implementation

The first component of our system is a compiler for the new language. It works much like a traditional compiler, accepting text files containing programs written in the appropriate syntax and producing an executable as output. When using the compiler directly, the user could hypothetically use any speech recognition system to dictate his code. However, this gives up much of the ability to influence the speech recognizer through programming context. In order to regain this, we need an entire spoken programming environment.

To create a complete programming environment, we created a plugin for the popular Eclipse IDE [12]. Our plugin uses the CMU Sphinx [13] speech recognition system to allow the user to dictate directly into a text editor within Eclipse. As the programmer dictates her program, background tasks analyze the code and update the speech recognizer to incorporate the ongoing context. Once the user is ready to compile, the plugin also integrates the Eclipse build system with our compiler to produce complete executables in the form of jar files that can run directly on a standard Java JVM.

To test the effectiveness of this system, we performed a small user study. Our users had varying levels of experience programming with a variety of other languages, but all of them had at least some basic programming knowledge. We found that users were able to begin dictating original programs with our new syntax with less than an hour of training. We observed informally that dictation speed and accuracy are much higher than reported by the spoken programming projects mentioned previously. Unfortunately, the source code for these earlier projects was not available at the time of our study, so we were unable to make a direct comparison.

More details about our system's performance and evaluation can be found in section 5, and a full description of the system implementation, baseline performance, and how it compares to other methods of spoken programming can be found elsewhere [1]. In the following section, we will describe the types of programming context that were considered and how they were integrated into the programming environment.

4. Adding Context

Next, we describe the improvements that were achieved by incorporating more programming context into the speech recognizer. The idea that context is important to understanding has been a part of AI and speech recognition for decades. Having a baseline spoken programming language, we wished to determine if the same idea would apply here. The two types of additional context we tested were scoping and type inference.

4.1. Scoping

Scoping is a very common idea in programming languages; it simply means that at any particular point in the program text, certain names (variables, functions, etc.) are visible, while others are not visible. For a statically scoped language like ours, the scope of a name corresponds to its position within the text. Typically, nested scopes are permitted, with names defined in inner scopes hiding symbols of the same name defined in outer scopes. Our language follows this typical paradigm through the use of nested blocks.

Knowledge of this structure can be exploited by the speech recognizer to improve its accuracy by favoring symbol names that are in scope. In this basic form, changing the probability of variable names based on scopes is similar to what many text-based IDEs already do during code completion. The major difference is that when speaking, the speaker does not say a partial word and then choose from a list of completions. Instead, the speaker says an entire utterance and expects the recognizer to choose the best option. As an example of how this might apply in practice, consider the partial program in Listing 1.

Listing 1: Scoping Example

```
set x to 0
```

```
while x < 10 do
    set y to x + 1
    print *
```

At the point marked $*$, imagine the user says something that sounds like "eye." What symbol should the recognizer insert? Going on pronunciation alone, potential English words might be "i", "eye", or "aye." However, a human following along with this conversation would easily realize that the intended word is "y." By incorporating scoping, we can give the speech recognizer this same knowledge.

In our plugin, this knowledge of scoping is implemented by splitting the Sphinx grammars that refer to variables into multiple grammar rules. Each time a new scope is entered, the existing grammar rules that refer to variable names are pushed onto a stack. When a new variable is brought into scope, any rules that recognize variable names are modified to include the new variable as a high-probability word choice. From that point through the end of that variable's scope, the speech recognizer will be biased in favor of using that variable name in appropriate slots. When that variable goes out of scope at the end of the block, the stack is popped so that the previous grammar rules are restored.

This mechanism also implements a second scope-related heuristic based on the principle of locality of reference. Much like a compiler uses this idea to generate more efficient code, the recognizer can guess that a person is more likely to refer to a recently-used or nearby variable than to a variable defined far away. Listing 2 contains a short example to illustrate this idea.

Listing 2: Locality of Reference Example

```
set i to the string Hello
set x to 0
while x < 10 do
    set y to x + 1
    if * = 5 then
```

Again, imagine that the user's utterance at the point labeled $*$ sounds like "eye." Unlike the previous example, both "i" and "y" are valid choices ("eye" and "aye" are still out of scope and therefore unlikely). However, y in this case seems like the more likely variable that was meant to be used. Because the scope containing y is at the top of the grammar rule stack, the extension from using scoping to favoring closer scopes is straightforward.

One thing that might stand out in the previous example is that there appears to be no way for the user to indicate that they really meant "i" when they said "eye." One mechanism for improving this difficulty is discussed in the next section.

4.2. Type Inference

The next type of context we added beyond scoping is type inference. In languages that use type inference, the types of symbols are typically still statically chosen at compile time, unlike dynamic languages. However, the compiler automatically chooses types based on the observed operations that are performed, freeing the programmer from much of the drudgery of declaring or changing variable types.

4.2.1. Motivation

Referring back to Listing 2, we can start to see how this can provide another source of context for the recognizer. When the user says “eye” at the point marked $*$, we can tell that the variable needed here should be something that can be compared to a number. Since i has been assigned a string, we can infer that its type is not numeric. For y, we can reason that x is a number and y has been assigned to something that is added to x, so it must also be numeric. This is just what is needed to fit in the slot, so “y” is the most likely word the user is trying to speak.

Because type inference provides additional restrictions on which variables can fit into a particular spot in the source code, it provides a natural override for the locality of reference heuristic described above. If a variable can’t be used in an expression then it doesn’t matter how local it is.

The program in Listing 2 actually provides a second place where type inference is useful. We have been claiming that the sounds for “i” and “y” are quite similar, so how does the recognizer know to insert a “y” when the user speaks line 4? The answer is partially that the user may pronounce the two sounds with extra care, of course, but it also lies in the fact that the two variables have different types. When i is created on line 1, it starts off as a string type. When line 4 is spoken, the user is attempting to assign a numeric type to some variable. Since the language uses static typing, the type of i can’t change in the middle of a function, which means the user must expect a new variable (or they’re creating a bug). Thus, because the recognizer has information about the inferred types of variables, it can conclude that the user is more likely to want “y” than “i” in the context of line 4.

4.2.2. Implementation

As mentioned in section 4.1, our programming environment does not just have one grammar rule for matching variable names. It has a rule for each scope, and the scope’s rule gets modified as additional variables come into scope. To pass type inference information to the speech recognizer, we extended this idea by splitting the rules again. Instead of a single rule that matches variables everywhere they might be needed within a scope, we have one rule that matches numeric variables, one for string variables, etc. The grammar rule for arithmetic expressions then refers to the numeric variables only, and so on. These rules are updated as type inference is performed so that the recognizer incorporates the current state of the program in progress.

While our compiler performs full type checking, we found this to be too slow for use in the recognizer. Instead, we perform “light” type checking by watching for patterns of use and marking variables with their probable types. A variable may be simultaneously marked with multiple probable types if it is used in contexts that provide insufficient information on their own. For example, the statement `read X` reads a number or string from the user, but this line alone does not indicate the type of X, so X is marked as both a probable string and a probable number. The recognizer will then accept X in contexts that require either type. When the full compiler is run, the type of X will either be fully determined, or any incompatible mixed uses will produce an error.

5. Evaluation

All of this description of how and why the language works is very nice, but how can its effectiveness be measured? This is a challenging question for any programming language, because most languages are judged on subjective measures like readability, expressiveness, consistency, etc. Even developing the "sense" of a programming language enough to form opinions on these measures takes time and practice with a language. Moreover, we do not want to evaluate the language in isolation; we want to evaluate the language together with its spoken programming environment. Speech recognition systems have their own evaluation challenges even when the phrases being recognized are meant to be things like common English sentences or simple commands. In this section, we describe how we came up with a repeatable mechanism that produces quantifiable results for evaluation.

5.1. Making Human Interaction Reproducible

Since we are dealing with human interaction, the first natural suggestion for evaluating the system is to conduct a user study. This is, in fact, what we did. There are several variations that we would ideally like to compare: the performance of our programming environment with the base recognizer, the performance after adding scoping, and the performance with both scoping and type inference turned on (type inference requires scoping, so evaluating type inference without scoping would be meaningless). The traditional way to do this might be to divide participants into groups and give each group the same task with a different version of the environment. The drawback to this approach is that it requires a large number of participants, and we did not anticipate finding enough volunteers.

Another approach is to ask each volunteer to use the system multiple times, e.g., once for each variation of interest. This allows for a smaller number of participants, but unfortunately introduces confounding training effects. For example, if a participant performs better on their third session, is that because the type inference is helping them, or have they merely gotten better with practice? In addition, it is very difficult to control for factors like ambient noise, distance between the speaker and the microphone, speech timing, etc.

We attempted to extract the best of each approach by recording and re-using the speech input from our participants. We experimentally verified that Sphinx's behavior when decoding files was deterministic, i.e., feeding the same audio input to Sphinx multiple times produced consistent output as long as the configuration was unchanged. We then modified the Eclipse plugin to accept audio input from a file instead of directly recording from the microphone. Thus, a single recorded input could be reused multiple times to simulate additional sessions on the same program text. Any differences in the textual output could be attributed with confidence to changes in the recognizer rather than to environmental effects.

An additional step we took to improve the reproducibility of our results was to ask participants to spend part of their session reading pre-written programs instead of composing on the fly. This ensured that participants were not stumbling over syntax issues or uncertainty about how to solve the problem. It also has the advantage that the pre-written program has a canonical correct output, namely, the original program text. This eliminates the common step of having a human carefully listen to and transcribe the person's audio to produce the expected textual output.

Finally, to further reduce any disfluences, we asked each participant to read each program twice, then discarded the first recording. All of this resulted in a set of audio files that we used to generate the textual output by passing it to the programming environment as described above. We then hand-classified each difference between the generated output and the original program text based on several categories that are explained in the next section.

5.2. Creating Correct Programs

Within the framework established by our study setup, we were able to test the effect of each variation on several potential measures of program correctness. We did not measure the commonly-used word error rate (WER) primarily because in our environment, errors are not independent. For example, if the recognizer misses the beginning of a **while** statement, the grammar will constrain it to also miss the **do** that starts the contained block and the **end while** that terminates the whole construct. Thus, a single mistake would be reported as at least four errors. Moreover, it is possible that other constructs might not be syntactically valid without the **while** loop, and might thereby add even more errors.

Instead, we came up with several measurements that we believe more accurately reflect the task of generating a program. We measured the following:

- Correct block structure
- Compilable program
- Correct block structure and compilable program
- Full functional equivalence

To explain each of these, we refer to the following short program that calculates the factorial of a non-negative integer:

Listing 3: Factorial

```
 1  define function f taking arguments n as
 2      if n < 2 then
 3          return 1
 4      else
 5          set r to 1
 6          set z to 2
 7          while z <= n do
 8              set r to r * z
 9              set z to z + 1
10          end while
11          return r
12      end if
13  end function
```

5.2.1. Correct Block Structure

Correct block structure is the idea that the output program contains the same blocks nested in the same manner as the intended input program. In Listing 3, an output program must contain a `while` loop inside the `else` block of an `if-else` statement inside a function. With the other statements stripped and nesting indicated by indentation, it looks something like this:

```
function
    if
    else
        while
```

If one of these levels is missed, an extra level is added, or they are nested incorrectly, the result does not have the correct block structure. The block structure indicates whether the overall outline of the program is correct. The program still may contain errors (and may not even compile), but it at least contains the core of the algorithm the user intended.

5.2.2. Compilable

Another measure of code correctness is a compilable program. This means the compiler is able to produce an executable from the input. The input must not contain syntax errors or erroneous constructs. However, it does not mean that the code is bug free. In fact, it doesn't even mean that the block structure is correct. For example, if the three parts of the `if-else` block are removed from lines 2, 4, and 12 in Listing 3, the block structure changes to this:

```
function
    while
```

This is clearly a different program, but it does still compile. Of course, the resulting function will not return the correct factorials of any numbers except 0 and 1.

5.2.3. Structure and Compilable

A much higher level of program correctness combines the previous two measures: A compilable program that also contains the correct block structure is getting close to a correct program. At this point, the program must not contain any syntax errors or invalid constructs. Errors found at this point are often going to be relatively simple bugs rather than unrecognizable changes to the code. For example, the `2` in line 2 could be substituted with another value without preventing compilation. An observer might even argue that it was essentially the same algorithm, but the output would be incorrect (i.e., a bug).

5.2.4. Functional Equivalence

The final measure of program correctness that we evaluated is full functional equivalence. With this measure, the program will compile and run, and will produce the same output as the original intended code. However, even at this level, we permit minor textual variations that do not change the program behavior. For example, if `z` in lines 6–9 of Listing 3 were uniformly replaced with `c`, the program would still perform the same computations and produce the same output. We would describe it as functionally equivalent rather than identical.

With a good set of unit tests, intuition suggests that it would be possible to check this level automatically by feeding a set of test data to the canonical program and the test program and comparing the two outputs. However, we did not perform this automatic verification because sometimes a different program structure can still produce identical output. This wrinkle is illustrated in our test program: Because of the structure of the function, the `else` block is actually unnecessary. That is, these two block structures are equivalent for this specific function:

```
function                function
    if                      if
    else                    while
        while
```

Because we want to measure whether the programming environment generates the source code the user is speaking, we classified this situation as a failure even though the program output meets the definition of functional equivalence.

5.3. Study Design

Putting together everything from the previous two sections, we describe the full study here.

5.3.1. Participant Demographics

We started with thirteen initial participants, but one was removed because his accent differed from General American by too much for Sphinx to produce usable results. Of the twelve subjects who were included in the final results, ten were male and two were female. Two had prior experience with voice recognition software, but ten reported having never used it. Two participants were self-described as novice programmers, five as intermediate, and five as expert. Two participants were undergraduates, two were non-students, and eight were graduate students.

5.3.2. Non-Interactive Reading

Each participant came in for a single recording session that lasted approximately an hour. The session was divided into three parts, made up of three different ways of interacting with the programming environment. During the first part of each participant's session, they were asked to read ten pre-written programs as described in section 5.1. During this part, participants were simply reading into a microphone; they were not interacting with the programming environment.

The programs themselves were structured to ensure that every syntactical feature of the language was exercised. Further, many of the programs purposefully included variable names that differed only in sound patterns that are often difficult for speech recognition engines to distinguish (e.g., n versus m or s versus f).

We took the recordings and used our programming environment to generate three text outputs from each audio file: one with the base recognizer, one with scoping enabled, and one with both scoping and type inference enabled. We then compared each text file to the corresponding original source code and hand-classified the differences according to the scheme described in the previous section. Finally, we used the Wilcoxon signed-rank test to evaluate the results for statistical significance. The results are described in section 6.1.

5.3.3. Interactive Reading

The audio files generated this way are excellent for isolating and measuring the results of changes to the programming environment. Unfortunately, they do not do much to measure actual user interactions with the software. To address this, the second part of the recording session involved re-reading the same ten programs a third time, but this time using the programming environment "live." Participants were shown how to make basic corrections and asked to attempt to reproduce the pre-written programs directly into the programming environment.

The results of this part are much less robust than the first part, due to all the factors described in section 5.1. Nevertheless, we did collect and analyze some basic information, such as successful completions and corrections required. These results are discussed in section 6.2.

5.3.4. Live Programming

The final test, of course, is to attempt live programming. One hour is not very long to learn a new programming language, and many of our participants were not necessarily strong programmers to begin with. Nevertheless, for those participants who had time left after the first two parts, we offered them three very simple challenge problems and let them try to use the live programming environment to write programs that solved the challenges. Because this last part was just for fun, we did not attempt to measure their performance, but we have recorded a few anecdotal observations in section 6.3.

6. Results

We next present the results of our user study on the spoken programming environment. We present non-interactive pre-written program reading in section 6.1, interactive program reading in section 6.2, and spontaneous interactive programming in section 6.3.

6.1. Results For Pre-written Programs

In this section, we present the results for each of the four metrics that we calculated on the pre-written program reading section of the user study. Each cell in the table shows how many programs meet the criteria for that particular metric. There were 12 total participants and 10 programs, so the maximum score in each cell is 10.

From the raw scores, we used the Wilcoxon signed-rank test to determine whether adding scoping and/or type inference produced a statistically significant improvement. We chose the Wilcoxon test instead of the common t-test because we cannot assume users' performance is normally distributed. Different users will have different disfluencies, speech speeds, etc., and even the same user will vary across different programs.

Because we are testing 12 hypothesis from the same data (3 combinations of context on 4 different tasks), we have applied a Bonferroni correction and will use $p = 0.004$ as the threshold of significance instead of the usual $p = 0.05$. More advanced corrections, such as the Benjamini-Hochberg method, would have given more power but would not have changed the final decision about which results are significant.

6.1.1. Correct Block Structure

Table 1 shows the raw counts for successful block structure. The majority of subjects experienced no change in the number of programs with correct block structure as a result of the context-based features. Of those who saw a difference, some were positive and some negative. However, the differences are not statistically significant. Using Wilcoxon's test, we have $W = 71.5$ with a p-value of 0.52 moving from the baseline to scoped context. Adding typed context to scoping, we have $W = 76, p = 0.42$. Finally, going from baseline to fully typed, we have $W = 75, p = 0.44$. This result is not much of a surprise, since the context-based improvements we have been considering should affect identifier recognition much more than the overall program structure or basic statement constructs.

Subject	Baseline	Scoped	Typed
1	5	5	4
2	6	6	6
3	8	8	7
4	8	8	9
5	9	8	9
6	8	8	8
7	10	10	10
8	7	8	8
9	9	9	9
10	10	10	10
11	8	8	8
12	8	8	8

Table 1: Block Structure Successfully Produced in Contexts

Subject	Baseline	Scoped	Typed
1	3	4	6
2	5	5	6
3	3	6	7
4	6	6	10
5	3	7	7
6	5	7	8
7	8	8	10
8	5	5	7
9	4	6	9
10	4	6	10
11	5	6	8
12	3	6	9

Table 2: Compilable Programs Produced in Contexts

6.1.2. Compilable Programs

The next test is for compilable programs. Raw counts are shown in Table 2. Compared to the block structure test, we should expect to see a larger effect for compilable programs from the context-based features. Errors such as using undefined variables or calling non-existent functions should be reduced or eliminated by the scoping, and we can expect errors such as adding a string to a number to be similarly reduced by the addition of typing.

Unlike the block structure test, no users experienced a decrease when adding context-based features. Some had an increase in compilable programs when adding scoping, some when adding typing, and some in both cases. Intuitively, this tells us that the context-based features have the expected effect, but formally we can check using Wilcoxon's test again. We find that $W = 117, p = 0.0039$ for adding scoping to the baseline. Adding typing to scoping, we get

$W = 125, p = 9.4 \times 10^{-4}$. Finally, going from baseline to fully typed, we get $W = 137, p = 8.7 \times 10^{-5}$. These p-values indicate very strongly that the increase in the number of compilable programs is not due to chance. The scoping and typing are individually statistically significant improvements, and the combination even more so.

6.1.3. Compilable and Correct Block Structure

Subject	Baseline	Scoped	Typed
1	2	3	3
2	5	5	6
3	3	6	7
4	6	6	9
5	3	7	7
6	5	7	8
7	8	8	10
8	5	5	7
9	4	6	8
10	4	6	10
11	5	6	8
12	3	6	8

Table 3: Correctly Structured Compilable Programs in Context

Next, we consider the combination of correct block structure and compilable program output. The raw counts shown in Table 3 are quite promising. Even at a glance, we can see that there are improvements for every subject and no regressions, so the context-based features appear to be a clear improvement. Performing our usual Wilcoxon test to verify this formally, we find $W = 115, p = 0.006$ for adding scoping to the baseline, not quite significant at our threshold of 0.004. Adding typing to this, we get $W = 118.5, p = 0.0033$. And finally, going from the baseline to full typing, we get $W = 128, p = 6.0 \times 10^{-4}$. In this case, scoping did not produce a statistically significant improvement by itself, but the combination of scoping and typing did.

6.1.4. Functional Equivalence

Finally, we have our strongest test: full functional equivalence. Table 4 shows the raw counts. Unfortunately, the results here are not quite so clear-cut. Eight out of 12 participants did show improvements with the context-based features turned on, but the results are not statistically significant. Adding scoping to the baseline, we get $W = 92.5, p = 0.12$. Adding typing, we get $W = 78, p = 0.37$, even less strongly supported. Even going from the baseline to fully typed, we get $W = 96.5, p = 0.08$. These all come up short of the normal 0.05 bar of statistical significance, let alone our lowered threshold of 0.004.

Full functional equivalence is obviously the most desirable outcome from our system, so the lack of a significant improvement here is disappointing. However,

Subject	Baseline	Scoped	Typed
1	1	2	1
2	3	2	2
3	3	4	4
4	5	6	7
5	2	4	4
6	4	5	5
7	8	7	9
8	5	5	5
9	3	3	3
10	3	5	5
11	3	3	4
12	2	4	5

Table 4: Functionally Equivalent Programs Produced in Contexts

this is actually where the limitations of re-using the pre-recorded offline data appear. Once interactivity is added, our results do show significant improvement due to the context-based features, as we will see in the next section.

6.2. Results For Interactive Reading

The second section of the user session was the interactive tests. As described in section 5.3, the user was directly interacting with the running programming environment for this part of their session. They were able—and encouraged—to make corrections when the system produced incorrect output. This means that unlike the offline recordings described in the previous section, the interactive sessions cannot be replayed through the system with different settings later.

Since we could not replay each particpants's recording multiple times to evaluate the context-based features, we randomly divided the participants into two groups. Of our twelve participants, six used the baseline system (group 1) and six used the full system with scoping and typing (group 2). Because of the small size of our study, we did not have any participants record with scoping only. Due to a recording error, interactive data for two of the participants was unusable; thus, we have five people in each group. Within each group, we have numbered the subjects from 1–5 without regard to their numbering in the offline section; thus, subject 1 in group 1 does not necessarily correspond to subject 1 from the offline tests. More importantly, subject 1 in group 1 is not the same person as subject 1 in group 2. The tables below have been divided by a vertical space to help visually reinforce this warning.

Because each participant was encouraged to make corrections as needed, we did not record whether the programs were perfect. For each participant, we recorded only whether they were able to complete each program within a 5-minute time limit. For completed programs, we also recorded how long it took and how many corrections were necessary.

Since we are comparing separate groups of people instead of comparing each participant to themselves, we cannot use the Wilcoxon signed-rank test (the Wilcoxon test requires that the data pairs come from the same population). In this scenario, the Mann-Whitney U test [14] is more appropriate for determining if the performance difference between the two groups is significant, so this will be the statistical test used in the following sections. Since we test three hypotheses from the same set of recordings, we have applied a Bonferroni correction and set our significance threshold to $p = 0.016$.

6.2.1. Completed Programs

Baseline Group	Completed
1	2
2	4
3	3
4	2
5	4

Scoped and Typed Group	Completed
1	8
2	10
3	10
4	10
5	10

Table 5: Programs Completed in Interactive Session

The first measure was the number of programs completed. None of the participants in the baseline group were able to complete more than 4 out of 10 programs within their 5 minute time limit. All of the participants in group 2 completed at least 8 out of 10 programs within 5 minutes, and 4 out of the 5 completed all 10 successfully. From this description alone, it is clear that group 2 was more successful than group 1. However, to present the result formally, we computed the Mann-Whitney test and found that $U = 25, p = 0.0046$. The data is shown in Table 5.

6.2.2. Corrections Needed

In addition to the raw number of programs completed, it is interesting to ask whether the number of corrections necessary to complete the program changed with the addition of the context-based features. The programs were different lengths, so we cannot simply compare the number of corrections directly. Instead, we have normalized by computing a “corrections per line of code” correction rate. In Table 6, we present the average correction rate for participants over all programs that each participant completed.

We did not include programs that were not completed in the calculation because it is impossible to know how many corrections might have been needed to finish whatever part remained. We observed that for incomplete programs, subjects had typically accumulated dozens of corrections before they either gave up or ran out of time. Had all of these corrections been included, the correction rates shown here for the baseline subjects would have been at least an order of magnitude larger.

Performing the Mann-Whitney test, we find that $U = 5, p = 0.07$. Thus, this result was not statistically significant. We can see from Table 6 that the range of

Baseline Group	Rate (err/line)
1	0.17
2	0.73
3	0.44
4	2.31
5	0.26

Scoped and Typed Group	Rate (err/line)
1	0.25
2	0.32
3	0.21
4	0.20
5	0.14

Table 6: Average Correction Rate Interactive Session

correction rates was significantly reduced by the addition of the context-based features (0.18 versus 2.14), but we cannot say that the overall distribution is definitely improved.

However, averaging in this way causes different numbers of programs to be included for each participant. To determine if the rates were affected by the inclusion of the extra programs for group 2, we performed the same calculation on a program-by-program basis for programs 1–3 (since these were completed by most participants). Even comparing indivdual programs directly to each other, we found no statistically significant differences in the rates of corrections needed.

6.2.3. Completion Time

The final comparison we performed between the groups was to see if the context-based features reduce the time needed to dictate a program. Like the corrections, we cannot directly compare the times. Not only are the programs different lengths, but different users have different natural speech rates, different pause lengths, etc. We normalized the times by dividing the seconds needed to complete the program by the number of lines to yield a seconds per line dictation rate. We averaged the dictation rates across the completed programs, and the results are shown in Table 7.

Baseline Group	Rate (sec/line)
1	4.8
2	7.14
3	3.86
4	30.0
5	6.33

Scoped and Typed Group	Rate (sec/line)
1	12.20
2	9.96
3	8.77
4	8.54
5	4.87

Table 7: Average Dictation Rates Interactive Session

Calculating our Mann-Whitney test, we find that $U = 18, p = 0.16$. Thus, we see the same picture as section 6.2.2: The range of dictation rates is tightened up considerably with the context-based features in group 2, but the overall distribution is not improved in a statistically significant way.

As in the previous section, we repeated the analysis of programs 1–3 for the dictation rate measurement to determine if the individual results differed from the average. We found no statistically significant differences at the individual program level here, either.

6.2.4. Discussion

Initially, these results seem counterintuitive. If the new context features do not necessarily reduce errors, and they do not necessarily let users dictate lines faster, how do they produce such a dramatic improvement in completed programs? We believe this is due to a combination of two factors. First, there is a large variability in user speech rates and enunciation. In our sample, this is most clearly seen in the difference between baseline subject 1 and baseline subject 4. Poor subject 4 took 6 times as long to dictate each line and made 13 times as many errors. The context-based features damp down this variability, but do not eliminate it entirely.

The other cause here is the 5-minute time limit that was used in the study. As mentioned above, for the programs that were not completed within five minutes, users had often accumulated dozens of errors. If we had allowed these users extra time to finish their programs, they would have then been counted into the correction and dictation rates. This would have affected mostly the baseline group, which would have pushed up their error rates and pushed down their dictation speeds. This may have been enough to tip it over into statistical significance.

6.3. Results For Challenge Problems

The third section of the user study was the programming challenges. In this section, we offered the user three small programming challenges, and they attempted to write programs to solve one or more of these challenges. Each of the challenges could be solved with less than 20 lines of code.

Here is an example challenge (introduction, samples, and hints elided):

> Read a number n from the user. Create an array that has $2x$ in each element x from 1 to n. Print the result.

A solution that follows the prompt in the most direct way possible might look something like this:

```
define function main taking no arguments as
    read n
    set element 1 of a to 0
    set x to 1
    while x <= n do
        set element x of a to 2 * x
    end while
    print a
end function
```

We gave the users freedom to solve the challenges in any way they liked, and every user had a different approach. Some users thought things through before they started dictating, while others started dictating immediately and then paused to think after every couple of lines. Some users had many more lines than others. Some users wrote almost exactly the solution we had expected, while others came up with something entirely different.

When counting a successful solution, we simply accepted whatever program was produced when the user stopped and told us they thought it was solved. If they stopped short, we counted that as a failure. Some of the programs actually were solutions to the problems given, but others contained logic errors or compilation problems that went unnoticed by the programmer. This seems no different than what one would expect if asking people to use a keyboard instead of dictating a program.

We observed that the problems were split between difficulties with using the new language correctly and errors caused by the spoken programming environment. The users who seemed more comfortable with the language after the first two sections had little trouble dictating programs to solve the challenges. Their errors were largely limited to incorrect variable substitutions, literal values, etc., similar to the errors observed in the offline testing. Those users who seemed less comfortable with the language syntax had significantly more trouble solving the challenges, but it was not necessarily because the spoken environment was making errors. In many cases, we observed that they were failing to dictate certain constructs because they were speaking incorrect syntax (e.g., "set function" instead of "define function" or "x equals 5" instead of "set x to 5"). Based on our observation, these users likely would have done much better with more practice. Since we did not control for training effects or attempt to ensure that users fully understood the spoken syntax prior to giving them the challenge programs, we did not investigate this possibility further.

One user did not attempt any challenges due to running out of time during the first two sections of the session. The other users all attempted at least one challenge. Six users attempted two problems, and one subject attempted all three.

One user did not solve any challenge problems in spite of attempting one, while the others who tried one or more solved at least one. Three of the six users who attempted two problems solved both, and the subject who attempted all three solved all three.

We think this is likely to be another manifestation of the situation described above, i.e., the people who felt confident in the language syntax had less trouble with the first challenge, and the positive feedback from completing it successfully made them more willing to attempt a second or even a third challenge. Conversely, the people who struggled to complete the first challenge program were unmotivated to try any additional spoken programming.

7. Conclusions and Future Work

In this paper, we presented a new programming environment that optimizes for spoken programming. Building upon the base idea of replacing the typical program syntax with the use of full words and English-like patterns, we described how further improvements in accuracy can be achieved by incorporating additional context-based knowledge into the recognizer. This merges some of the strengths of existing English speech recognizers with the domain-specific knowledge and regular structure that come from a programming language.

To evaluate this environment, we completed a user study of our programming environment in which we analyzed the performance gains contributed by our added forms of contextual awareness. We found strong statistically significant improvements in participants' ability to create correctly structured, compilable programs. This was especially true when using the system interactively: All participants who used the context-based features were able to complete more programs than any participants who did not have the context-based features enabled. This result was statistically significant with $p = 0.0046$.

Taken all together, this research paints a picture of a possible future for spoken programming. Lacking physical keyboards, tablets are currently unsuited for major programming tasks, but being able to enter programs via speech eliminates one major obstacle to their use for that purpose. Even given the idea of speaking programs, speaking C, Java, or other traditional computer languages is likely to run into similar limitations to those found in previous research. Rather than attempting to turn human speech into existing computer languages, we have demonstrated that significantly better spoken results can be achieved by molding the computer language to human speech patterns.

Of course, before our new spoken language could be a serious alternative to Java or C, plenty of further research is needed. There are many possible avenues for further study in this area, but we will only mention a few here.

For spoken programming to be successful, the initial hurdle of remembering syntax and achieving successful program entry must be overcome. However, once that is in place, most programmers spend more time editing and debugging than on the initial program entry, so spoken source code navigation, editing, and debugging are vital areas for future study.

Spoken navigation is also an area ripe for consideration of multi-modal interfaces. Saying "go up three lines" may be intuitive, but "select this line" with an accompanying screen touch is probably even more so. Once touch is incorporated, many selection or movement tasks can more easily be described with a gesture than with a wordy description. One can easily imagine the user circling or tapping variables, lines, or whole functions of interest to tell the editor where to make their spoken changes.

The final area of future research we mention here is higher-level language constructs. The language we developed is around the level of C, with a few additions like automatic memory management and type inference. However, most programming languages today supply significantly more built-in functionality. When a language is Turing complete, it might be argued that all of these features are merely syntactic sugar, but if that made no difference, people would never have developed anything beyond C. The syntactic sugar is not merely pleasant; in many cases it is a core part of a language and a source of programmer productivity. We suggest that spoken syntax for more advanced type systems and object-oriented programming represents an important next step.

With this project, we have taken a small step in the direction of a new way to think about programming environment design: speech first instead of speech as an afterthought. Tablets and similar devices are here to stay; if programmers aren't already asking to program on their tablets, they will be soon. Giving programmers a bluetooth keyboard and a Java compiler is unlikely to be a satisfactory answer for long. We hope that more small steps toward spoken programming will eventually lead to a large jump in the right direction.

[1] B. M. Gordon, Improving spoken programming through language design and the incorporation of dynamic context, Ph.D. thesis, University of New Mexico, Albuquerque, New Mexico (2013).

[2] J. L. Leopold, A. L. Ambler, Keyboardless visual programming using voice, handwriting, and gesture, in: 1997 IEEE Symposium Visual Languages, 1997, pp. 28–35. doi:10.1109/VL.1997.626555.

[3] A. Désilets, D. C. Fox, S. Norton, Voicecode: An innovative speech interface for programming-by-voice, in: CHI '06 Extended Abstracts on Human Factors in Computing Systems, CHI EA '06, ACM, New York, NY, USA, 2006, pp. 239–242. doi:10.1145/1125451.1125502.
URL `http://doi.acm.org/10.1145/1125451.1125502`

[4] A. Désilets, D. C. Fox, S. Norton, VoiceCode Programming by Voice Toolbox, `http://sourceforge.net/projects/voicecode/` (Retrieved August 30, 2014).

[5] A. Begel, S. L. Graham, Spoken Programs, in: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, 2005, pp. 99–106. doi:10.1109/VLHCC.2005.58.

[6] A. Begel, S. L. Graham, An Assessment of a Speech-Based Programming Environment, in: 2006 IEEE Symposium on Visual Languages and Human-Centric Computing. VL/HCC 2006., 2006, pp. 116–120. doi:10.1109/VLHCC.2006.9.

[7] S. C. Arnold, L. Mark, J. Goldthwaite, Programming by voice, VocalProgramming, in: Proceedings of the fourth international ACM conference on Assistive technologies, Assets '00, ACM, New York, NY, USA, 2000, pp. 149–155. doi:http://doi.acm.org/10.1145/354324.354362.
URL `http://doi.acm.org/10.1145/354324.354362`

[8] T. Rudd, Using Python to Code by Voice, `http://pyvideo.org/video/1735/using-python-to-code-by-voice` (Retrieved August 30, 2014).

[9] B. C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, USA, 2002, Ch. 22: Type Reconstruction, pp. 317–338.

[10] E. W. Dijkstra, Notes on structured programming, in: O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare (Eds.), Structured programming, Academic Press Ltd., London, UK, UK, 1972, Ch. 1, pp. 1–82.
URL `http://portal.acm.org/citation.cfm?id=1243380.1243381`

[11] A. J. Perlis, Epigrams on programming, SIGPLAN Notices 17 (9) (1982) 7–13.

[12] Eclipse - The Eclipse Foundation open source community website., `http://www.eclipse.org/` (Retrieved December 30, 2014).

[13] CMU Sphinx - Speech Recognition Toolkit, `http://cmusphinx.sourceforge.net/` (Retrieved September 1, 2014).

[14] N. Nachar, The mann-whitney u: A test for assessing whether two independent samples come from the same distribution, Tutorials in Quantitative Methods for Psychology 4 (1) (2008) 13–20. doi:10.20982/tqmp.04.1.p013.
URL `http://www.tqmp.org/RegularArticles/vol04-1/p013/p013.pdf`