Translating Keyword Commands into Executable Code

Greg Little & Robert C. Miller
MIT CSAIL
32 Vassar St
Cambridge, MA 02139 USA
{glittle,rcm}@mit.edu

ABSTRACT

Modern applications provide interfaces for scripting, but many users do not know how to write script commands. However, many users are familiar with the idea of entering keywords into a web search engine. Hence, if a user is familiar with the vocabulary of an application domain, we anticipate that they could write a set of keywords expressing a command in that domain. For instance, in the web browsing domain, a user might enter ***click search button***. We call expressions of this form *keyword commands*, and we present a novel approach for translating keyword commands directly into executable code. Our prototype of this system in the web browsing domain translates ***click search button*** into the Chickenfoot code ***click(findButton("search"))***. This code is then executed in the context of a web browser to carry out the effect. We also present an implementation of this system in the domain of Microsoft Word. A user study revealed that subjects could use keyword commands to successfully complete 90% of the web browsing tasks in our study without instructions or training. Conversely, we would expect users to complete close to 0% of the tasks if they had to guess the underlying JavaScript commands with no instructions or training.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces; D.3.3 [Programming Languages]: Language Constructs and Features; D.2.6 [Programming Environments]: Interactive environments; H.5.2 [User Interfaces]: User-centered design.

General terms: Algorithms, Design, Experimentation, Human Factors, Standardization, Languages.

Keywords: End-user programming, Command languages, Natural Language Processing, Web automation.

INTRODUCTION

Many modern applications have scripting interfaces. These interfaces are powerful tools, both for automating tasks within applications and for coordinating tasks between applications. Unfortunately, learning to write scripts is a prohibitive barrier for many users. Three factors contribute to the learning problem. First, users must learn to cope with rigid and seemingly arbitrary syntax rules for formulating expressions. The canonical example is the semicolon required at the end of each expression in C, but even modern scripting languages like Python and JavaScript have similar arbitrary syntax requirements. Second, users must often learn several different scripting languages and be able to switch between them. This is necessary because of the wide variety of languages in use by applications, and it is difficult because the scripting languages can be very similar, but different enough to cause problems. Finally, users must learn the Application Programmer Interface (API) for the application they want to script, but API's can be quite large and it can be difficult to isolate the portion of the API relevant to the current task. This last factor is also known as the *selection* barrier [13].

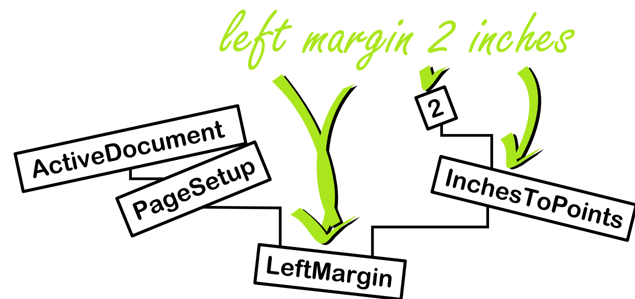

Figure 1: Illustration of keyword command translation.

Much end-user programming research has focused on these problems. Two notable approaches include programming-by-demonstration (PBD) and structured editors. The basic idea behind PBD (e.g. [7, 14]) is to infer a program from manual actions taken by the user, like a macro recorder. This approach can be a good first step toward creating a script, but it is often easier to correct problems and generalize the script using a script editor. Structured editors (e.g. [1, 10, 11, 17, 24]) can be a good approach for end users to write and edit scripts, since they force the script to retain a syntactically correct form. They also often provide menu systems so the user can recognize, rather than recall, commands. For example, Apple Automator [1] provides a menu listing the available application commands, and each command's parameters can be configured by a graphical user interface. However, structured editors tend to require the user to think in a certain order, which can be counter-intuitive. Also, structured editors sacrifice some of the benefits inherent to purely textual environments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST'06, October 15–18, 2006, Montreux, Switzerland..
Copyright 2006 ACM 1-59593-313-1/06/0010 ...$5.00.

The benefits of purely textual environments are too great to dismiss outright, even for end-user programming. Consider that text is ubiquitous in computer interfaces. Facilities for easily viewing, editing, copying, pasting, and exchanging text are available in virtually every user interface toolkit and application. Plain text is very amenable to editing—it is less brittle than structured solutions. Also, text can be easily shared with other people through a variety of communication media, including web pages, paper documents, instant messages, and e-mail. It can even be spoken over the phone. Tools for managing text are very mature. The benefits of textual languages for programming are well-understood by professional programmers, which is one reason why professional programming languages continue to use primarily textual representations.

The primary hurdle for end-users has been creating textual expressions that the computer will understand. Previous efforts in this regard have focused on natural language programming. Sammet advocated the use of English (or any natural language) as a programming language as early as 1966 [23]. However, interpreting arbitrary English expressions as executable code has proven to be a real challenge. A study by Miller [16] outlines some of these challenges, including the variety of styles humans use to express ideas.

Because of this, many natural language programming systems are built around grammars or templates which impose some constraints on input expressions. Grammar based systems like NLC [2] use formal languages that read like English. The problem with these systems is that they do not permit grammatical errors or extraneous words. Template based systems like NaturalJava [22] try to overcome these problems by searching for recognizable language constructs within an expression. However, these systems still require the user to incorporate such constructs into their expressions.

Our approach goes one step further by eliminating the need for language constructs all together, and focuses on the presence of keywords in a command expression. We call such expressions *keyword commands*. Consider the keyword command ***left margin 2 inches*** in the context of Microsoft Word. To a human reader, this suggests the command to make the left margin of the current document 2 inches wide. Such a command can be expressed in the formal language of Visual Basic as ***ActiveDocument.PageSetup.LeftMargin = InchesToPoints(2)***. A prototype of our system can make this translation automatically from the original keyword command. (see Figure 1).

Several key points to note are: First, the user did not need to worry about strict requirements for punctuation and grammar in their expression. For instance, they could have said something more verbose, like ***set the left margin to 2 inches***, or they could have expressed themselves in a different order with ***2 inches, margin left***. Second, the user did not need to know the syntactic conventions for method invocation and assignment in Visual Basic. The same keyword command would work regardless of the underlying scripting language. Finally, the user did not have to search through the API to find the exact name for the property they wanted to access. They also didn't need to know that ***LeftMargin*** was a property of the ***PageSetup*** object, or that this needed to be accessed via the ***ActiveDocument***.

One advantage of this approach over PBD is that it allows users to create scripts that access functionality they may not know how to access manually. For instance, assume a user wants to create 3 columns in their Word document as part of a script, but they do not know which dialog affords this change. They would need to discover the proper dialog before they could demonstrate the command to a PBD system. However, our approach would allow them to guess a keyword command like ***3 columns***, and the system would do the work of searching through the API to construct an expression likely to achieve this goal.

This example also illustrates a key drawback of structured editors. If the user wants to express the command ***3 columns*** in a structured editor, they need to begin by filling in the first slot in an expression builder, and this slot is unlikely to accept the number 3. In fact, they may not know whether to begin with an assignment template or a function template, depending on how this particular property is set in the API. The main point here is that Structured Editors require some planning on the part of the user toward building their expression, even if they can pick out the pieces with menu systems. Our approach relaxes this restriction.

Another advantage of keyword commands over programming-by-demonstration and structured editors is that it accommodates *pure text*. This affords all the benefits cited above, and also allows these expressions to serve as meta-URLs for bookmarking application states. One virtue of the URL is that it's a short piece of text—a command—that directs a web browser to a particular place. Because they are text, URLs are easy to share and store. Keyword commands offer the same ability for arbitrary applications—you can store or share a small set of keyword commands that will put an application into a particular state. On the web, this could be used for bookmarking *any* page, even if it requires a sequence of browsing actions. It could be used to give your assistant the specifications for a computer you want to buy, with a set of keyword commands that fill out forms on the vendor's site in the same way you did. In a word processor, it could be used to describe a conference paper template in a way that is independent of the word processor used (e.g. ***Arial 10 point font***, ***2 columns***, ***left margin 0.5 inches***).

To test our idea, we have implemented two prototypes. One prototype operates in the web browsing domain, and translates keyword commands into Chickenfoot [6] commands. These commands allow users to navigate to web pages and interact with web forms. The other prototype operates in the domain of Microsoft Word, and translates keyword commands into Visual Basic commands. These commands can be used to access document properties and issue commands like Save and Print.

We also conducted a user study of the web domain prototype to gauge the ability of end users to form keyword commands on their own, without instructions or training. We chose the web domain because end users were likely to be familiar with the capabilities of a web browser.

We found that users were able to generate successful keyword commands for 90% of the tasks, and that their first attempt succeeded 73% of the time.

The next section addresses related work. This is followed by a description of the user interface for keyword commands. After that, we discuss the implementation of the two keyword command prototypes. Next, we present the user study. Finally, we end with future work and concluding remarks.

RELATED WORK

Interest in *natural programming* was renewed recently by the work of Myers, Pane, and Ko [19], who have done a range of studies exploring how both non-programmers and programmers express ideas to computers. These seminal studies drove the design of the HANDS system, a programming environment for children that uses event-driven programming, a novel card-playing metaphor, and rich, built-in query and aggregation operators to better match the way non-programmers describe their problems. Event handling code in HANDS must still be written in a formal syntax, though it resembles natural language.

Bruckman's MooseCrossing [3] is another programming system aimed at children that uses formal syntax resembling natural language. In this case, the goal of the research was to study the ways that children help each other learn to program in a cooperative environment. Bruckman found that almost half of errors made by users were syntax errors, despite the similarity of the formal language to English [4].

More recently, Liu and Lieberman have used the seminal Pane & Myers studies of non-programmers to reexamine the possibilities of using natural language for programming, resulting in the Metafor system [15]. This system integrates natural language processing with a common-sense knowledge base, in order to generate "scaffolding" which can be used as a starting point for programming. Keyword commands also rely on a knowledge base, but representing just the application domain, rather than global common sense.

The keyword command implementation, which is essentially a search through the application's API, is similar to the approach taken by Mandelin et al. for *jungloids* [18]. A jungloid is a snippet of code that starts from an instance of one class and eventually generates an instance of another (e.g., from a File to an Image). A query consists of the desired input and output classes, and searches the API itself, as well as example client code, to discover jungloids that connect those classes. Keyword commands must also search the API to generate valid code, but the query is richer, since keywords from the command are also used to constrain the search.

Some other recent work in end-user programming has focused on understanding programming errors and debugging [12, 13, 20], studying problems faced by end-users in comprehension and generation of code [13, 25] and increasing the reliability of end-user programs using ideas from software engineering [5, 8]. Keyword commands do not address these issues, and may even force tradeoffs in some of them. For example, a keyword command program may be less reliable, as discussed in the *Limitations* section later in this paper.

USER INTERFACE

The heart of the user interface is keyword command generation. The user needs to conjure some command that the computer is likely to understand. Toward this end, the user must have some idea of the capabilities of the system, along with the conventional vocabulary used to describe these capabilities (e.g. the white-space surrounding a document is called the "margin"). Beyond this, our hope is that the system is natural and intuitive. That is, a user should not have to read the following sections in order to *use* keyword commands.

Functions

The user can invoke a function by including keywords in their command which are present in the name of the function. For instance, in the example ***left margin 2 inches***, the word ***inches*** appears in the function ***InchesToPoints***. A single word is sufficient to identify a function, even if the word appears in other functions, so long as only one function fits well into an interpretation of the entire command. The framework also allows functions to have synonyms, giving the user some slack in remembering the exact name of a function.

The user can also invoke a function without naming it, merely by including its arguments. This is a sufficient suggestion in those cases where there is only one function which is likely to take the given arguments. For instance, consider ***UIST 2006 into the search textbox***. This suggests entering "UIST 2006" into the textbox labeled "search", even though we didn't explicitly say ***enter***. Our web domain prototype understands this expression because it has only one command that accepts a string and a textbox (namely ***enter***).

Basic Data Types

Like functions, the user can also create basic data types by including keywords in their command which represent these types. For instance, the user can create the integer 2 with any of the keywords ***2***, ***two***, ***2nd***, ***second***, etc... depending on how many variations the interpreter has in its database.

Strings themselves are basic data types, and can be created by just including the words of the string in the command. However, a sequence of words is considered more likely to be a string if the user places quotes around it. We discuss this more below when we talk about resolving ambiguity.

The exact set of basic data types depends on the domain of the interpreter. Our web domain prototype includes types for integers, strings, booleans, URLs, and keyword-lists (which identify objects in the webpage). The Word domain prototype includes types for ints, longs, booleans, and strings.

Identifying Arguments

If a command takes multiple arguments of the same type, then the user may need to supply words in their expression to identify the arguments. One method is to name the arguments. Argument names can appear immediately before or after the words used to express the argument. Another method is to include the arguments in the correct order. For instance, if we had a ***click(integer x, integer y)*** command, then the system would translate both ***click 300 y 200 x*** and ***200 300 click*** into ***click(200, 300)***.

Prepositions can serve as intuitive names for some argu-

ments. For instance, the command ***copy(path toDestination, path fromSource)*** allows for expressions like ***copy A:\my_paper.doc to C:\my_backup***. In this case, the arguments are out of order, but the system picks up on the word ***to***, which is part of the name for the ***toDestination*** argument. In practice, we include many synonyms for these prepositions to support variations like ***copy A:\my_paper.doc into C:\my_backup***.

It is also possible for the system to disambiguate arguments of the same type based on domain specific heuristics. Continuing the example above, we could say ***A:\my_paper.doc C:\my_backup copy***, where the arguments are out of order, but we do not provide a preposition. In this case, the system can disambiguate the arguments using the heuristic that if only one argument is a directory, then it is the destination.

Nested Functions

The system supports nested functions. For instance, ***pick the 4GB RAM option*** translates into ***pick(findOption("4GB RAM"))***. Note the nested invocation of ***findOption***. The one restriction regarding nested functions is that a nested function can only be formed from a contiguous portion of the keyword command. This example obeys this restriction because ***4GB RAM option*** is a contiguous string within the command. We could have also exchanged the order to get ***option 4GB RAM***, but we could not split up the subexpression to get ***option pick 4GB RAM***.

In some cases, the system may make the correct translation, even if the subexpression is split. In the case of ***option pick 4GB RAM***, the system fails to find a use for ***findOption()*** as a nested function with no arguments, and so it favors an interpretation involving just the keywords ***pick 4GB RAM***. The interpreter then introduces the unnamed function ***findOption***, since the user supplied arguments suggesting this command, namely ***4GB RAM***.

Even without this fallback mechanism, this restriction does not appear to be prohibitive. No users in our study formed expressions violating this rule (and users correctly formed 31 expressions which could have violated the rule).

Resolving Ambiguity

Sometimes an expression needs to use words in an ambiguous way. The most common example is trying to quote text, when the text itself contains words with other likely interpretations. Consider ***enter binary search textbox***. Do we want to enter the word "binary" into the "search" textbox, or do we want to enter "binary search" into the only textbox on the page? The system supports a couple of techniques for resolving such ambiguities.

Quotes: The first technique is to include quotes. When the system considers the possibility that a sequence of words represents a string (or keyword-list), it gives this possibility more weight if the user includes quotes on either side of the sequence. We could therefore say ***enter "binary" search textbox***, and the system would favor an interpretation where ***"binary"*** was treated as a string separate from ***search***.

When using quotes, it is not necessary to place escape characters in front of quote symbols which are embedded within a quote. The system will resolve this ambiguity based on how many string arguments it requires to form a valid interpretation for the whole command. For instance, the expression ***enter "history of "foo bar"" into the search textbox*** resolves to ***enter("history of \"foo bar\"", findTextbox("search"))***, despite the nested quotes in the original expression.

Argument Names: Another technique is to include argument names to identify an argument, as discussed earlier. Using this technique we could say ***enter binary into search textbox***, and the system would favor an interpretation where the subexpression ***into search textbox*** was treated as the ***into*** argument of the ***enter*** command.

Selecting From List: When these techniques fail (or the user fails to employ them), the system is left with an ambiguous expression. In such a case, it can present the most likely candidates to the user for inspection. This is discussed more in the *Feedback* and *Graphical User Interface* sections below.

Extraneous Words

We explained that the system recognizes certain stop words in certain situations (e.g. ***to***, ***into***), but these are only recognized if the API designer has incorporated these words into argument names. Also, the user might include other extraneous words in their expressions. Consider ***please enter "search" into the textbox***. What is the system supposed to do with the word "please"? In this case, it ignores it— since no interpretation of the expression has a good explanation for the word, the system considers interpretations which simply overlook it. Note that the system will even overlook words which identify functions if they are incompatible with the rest of the expression.

Feedback

It is useful to let the user know what action the computer has taken in response to a command. One method is to supply a graphical indication. For instance, when a user clicks a link in a webpage using a keyword command, our web prototype animates a green translucent rectangle over the link. This assures the user that the correct link was chosen.

In some cases, it is not feasible to supply a graphical indicator. In these cases, it is useful to provide textual feedback. This feedback can come in the form of displaying the resulting code generated from the command. Our framework also provides a mechanism for generating pseudo-natural language representations for commands, which end-users may find easier to read. These interpretations may also act as guides for future expressions since they are themselves interpretable by the system. For instance, if the user enters the command ***300GB Hard Drive***, and the system translates this to ***select(findRadioButton("300GB Hard Drive"))***, then the pseudo-natural language feedback would look like ***select the "300GB Hard Drive" radiobutton***, which is also an expression understood by the system.

Textual feedback is also useful when the user enters an ambiguous command, and the system wants to afford the selection of an interpretation from a list. In these cases, it may be easier to represent the alternatives with text, rather than a graphical indication.

Figure 2: a) command box, b) feedback bar, c) animated acknowledgement

Graphical User Interface

Our web browser prototype consists of a textbox affording input, and an adjacent horizontal bar allocated for textual feedback. The system also generates an animated acknowledgement in the web page around the html object affected by a command (see Figure 2).

We plan to extend this system with a dropdown menu to present a list of likely interpretations for ambiguous commands (the current prototype makes an arbitrary choice in such cases). We also envision incorporating the system into a script editor as a form of auto-completion. For instance, if a user types a line of text in the Chickenfoot script editor that doesn't parse as JavaScript, the system could present valid JavaScript interpretations of the expression in an auto-completion style popup menu.

IMPLEMENTATION

This section describes how keyword commands are translated into executable code.

Functions

Functions are the building blocks in the system. Each function returns a single value of a certain type, and accepts a fixed number of arguments of certain types. Basic data types are represented as functions that take no arguments. We can implement optional arguments with function overloading.

Functions can have many names. For instance, the ***enter*** command in the web prototype has names like ***type***, ***write***, ***insert***, ***set***, and the = symbol. Functions for basic data types have names corresponding to their textual representations. These names are matched programmatically with regular expressions, e.g., integers are matched with "[0-9]+".

Arguments can also have many names, which may include prepositions naming the grammatical role of the argument. We can implement named arguments as functions which take one parameter, and return a special type that fits into their parent function.

Translation Algorithm

The translation algorithm needs to convert an input expression into a likely function tree. We describe it in two steps.

Step 1: Tokenize Input

Each sequence of contiguous letters forms a token, as does each sequence of contiguous digits. All other symbols (excluding white space) form single character tokens. Letter sequences are further subdivided on word boundaries using several techniques. First, the presence of a lower-case letter followed by an upper-case letter is assumed to mark a word boundary. For instance, ***LeftMargin*** is divided between the ***t*** and the ***M***. Second, words are passed through a spell checker, and common compound expressions are detected and split. For instance, ***login*** is split into ***log in***. Note that we apply this same procedure to all function names in the API, and we add the resulting tokens to the spelling dictionary.

One potential problem with this technique is that a user might know the full name of a property in the API and choose to represent it with all lower-case letters. For instance, a user could type ***leftmargin*** to refer to the ***LeftMargin*** property. In this case, the system would not know to split ***leftmargin*** into ***left margin*** to match the tokens generated from ***LeftMargin***.

To deal with this problem, the system adds all camel-case sequences that it encounters to the spelling dictionary before splitting them. In this example, the system would add ***LeftMargin*** to the spelling dictionary when we populate the dictionary with function names from the API. Now when the user enters ***leftmargin***, the spell checker corrects it to ***LeftMargin***, which is then split into ***Left Margin***.

After spelling correction and word subdivision, tokens are converted to all lower-case, and then passed through a common stemming algorithm [21].

Step 2: Recursive Algorithm

The input to the recursive algorithm is a token sequence and a desired return type. The result is a tree of function calls derived from the sequence that returns the desired type. This algorithm is called initially with the entire input sequence, and the desired return type ***void***, since we want the command to *do* something as opposed to return something.

The algorithm begins by considering every function that returns the desired type. For each function, it tries to find a substring of tokens that matches the name of the function. For every such match, it considers how many arguments the function requires. If it requires n arguments, then it enumerates all possible ways of dividing the remaining tokens into n substrings such that no substring is adjacent to an unassigned token. Then, for each set of n substrings, it considers every possible matching of the substrings to the n arguments. Now for each matching, it takes each substring/argument pair and calls this algorithm recursively, passing in the substring as the new sequence, and the argument type as the new desired return type.

The resulting function trees from these recursive calls are grafted as branches to a new tree with the current function as the root. The system then evaluates how well this new tree explains the token sequence (see *Explanatory Power* below). The system keeps track of the best tree it finds throughout this process, and returns it as the result.

The system also handles a couple of special-case situations:

Extraneous Tokens: If there are substrings left over after extracting the function name and arguments, then these substrings are ignored. However, they do subtract some explanatory power from the resulting tree.

Inferring Functions: If no tokens match any functions that return the proper type, then the system tries all of these functions again. This time, it does not try to find substrings of tokens matching the function names. Instead, it skips directly to the process of finding arguments for each function.

Of course, if a function returns the same type that it accepts as an argument, then this process can result in infinite recursion. We currently handle this by not inferring commands after a certain depth in the recursion.

Explanatory Power

Function trees are evaluated in terms of how well they explain the tokens in the sequence from which they are derived. Tokens are explained in various ways. For instance, a token matched with a function name is explained as invoking that function, and a token matched as part of a string is explained as helping create that string.

Different explanations are given different weights. For instance, a function name explanation for a token is given 1 point, whereas a string explanation is given 0 points (since strings can be created from any token). This is slightly better than tokens which are not explained at all—these subtract a small amount of explanatory power. Also, inferred functions subtract some explanatory power, depending on how common they are. Inferring common functions costs less than inferring uncommon functions. Currently, we hard code these values, but in the future they should reflect an a priori probability from a corpus of user data.

Web Prototype

The functions in the web prototype map to commands in Chickenfoot [6]. We include 18 functions, with an average of 6.4 synonymous names for each function. Many functions share some of the same names. For instance, ***make*** is a name for the ***pick*** function and the ***enter*** function. These ambiguities are resolved based on argument types.

Word Prototype

The prototype of the system in the domain of Microsoft Word presented some challenges which were not present in the web prototype. First, we were faced with a much larger command set. Chickenfoot has less than 20 commands, whereas the Word API has over two thousand. This meant we could not create all the functions by hand. Instead, we mined Word's type libraries, and converted its properties and methods into functions automatically. However, we were not able to populate the system with function synonyms.

We also had to turn off the ability of the system to infer commands which were not named in the expression if these commands required additional arguments, since this dramatically increased the search space. However, we were able to achieve some of the same benefits of inferring commands with a few modifications to the algorithm.

First, we wanted to be able to infer commands which returned useful objects in the system like ***ActiveDocument.PageSetup***, which returns an object containing properties for margin sizes. (Note that ***PageSetup*** is the command, and ***ActiveDocument*** is the argument.) We noticed that most of these objects were accessible as descendants of either ***ActiveDocument*** or ***ThisApplication***; hence, we implemented an algorithm to enumerate all such descendants, and store them in a list. We then allowed the system to infer commands in this list as if they took no arguments (since we kept track of which arguments were required to access each command).

The next modification is best introduced with an example. Consider the expression ***columns 2***. The desired interpretation is ***ActiveDocument.PageSetup.TextColumns.SetCount(2)***, but this requires inferring the command ***SetCount*** since no words from this command appear in the original expression. However, we note that ***SetCount*** has an argument named ***NumColumns***, and the word ***Columns*** *does* appear in the expression. Hence, in such cases, we mitigate the need to infer a command by including function argument names as synonyms for function names.

The final modification is also motivated by an example. Consider the expression ***A4***, which we want to translate to ***ActiveDocument.PageSetup.PaperSize = wdPaperA4***. In this case, we need to infer the command ***PaperSize***. The way we get around this is to allow the system to search for commands with any return type, instead of just commands returning ***void***. Then, if the return is not ***void***, we try to fit the return value into some other function that does return ***void***.

In our example, the system returns the function ***wdPaperA4***, which has the return type ***WdPaperSize***. The system then searches for a function which takes something of type ***WdPaperSize*** as an argument. Remarkably enough, there is only one such function, namely ***PaperSize***. This function also requires an argument of type ***PageSetup***, but we allow the system to fill in such requirements using the list we built earlier, which supplies the function ***ActiveDocument.PageSetup***.

Speed

We implemented the translation algorithm for each prototype in Java. The following running times provide a feel for the speed of this algorithm.

For the web prototype, we used inputs from the user study to derive an average parse time of 44 milliseconds (on an AMD Athlon 4200+ processor). Figure 3 shows how the parse time varies for input sizes of different lengths. From this we can guess that the average-case running time is polynomial, but that it is reasonable for inputs up to 8 tokens long (taking less than 300 milliseconds on average). Note that this seemed adequate for tasks in the user study since 96% of the user inputs were 8 tokens or less.

It is also worth looking at the worst-case running time to get a feel for how different factors affect the search space. Suppose the user's input has n tokens, and every substring of tokens matches f functions in the library, each taking a arguments. The first call to the recursive algorithm must try every way to divide the n tokens into $a + 1$ substrings in any order (one for each argument plus the function name it-

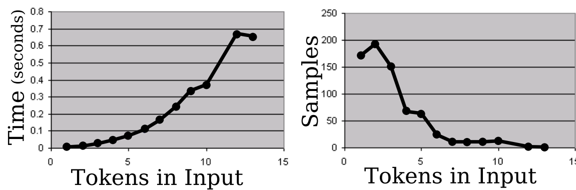

Figure 3: The left chart shows the time it took to parse inputs of various lengths, while the right chart shows how many samples we had of each length.

self), which is $\binom{n-1}{a}(a+1)! = O(an^a)$. And since each substring matches f functions, this gives $O(fan^a)$ possibilities for each recursive call. Since the function tree for n tokens can have at most n nodes (ignoring function inference), the total search time would be $O((fan^a)^n)$. In practice, f should be small (tokens match few functions), a is small (most functions take few arguments), and n is small (users use few tokens), so this worst case is unlikely to bite.

The Word prototype translates many short expressions (4 words or less) in less than a second, however the running time increases significantly for longer expressions. It also depends on the words used. Including multiple words which appear in many commands can dramatically increase the translation time. For instance, the expression ***left left*** takes 4 seconds. This already suggests room for optimization in the algorithm; for instance, making common words like ***left*** insufficient by themselves to invoke any commands.

Our overall conclusion is that the current algorithm is sufficient for small command sets (under 20 commands) where the user is unlikely to enter large expressions (over 8 words). However, we believe the algorithm already generates some very useful results in larger domains, and we want to explore this potential in future work.

Limitations

The correct interpretation of a keyword command depends on several factors: the weights and heuristics used for parsing; the set of functions available in the API; and the state of the application itself. Changes in these factors may change the way a command is interpreted. For example, typing ***refresh*** in the web domain would normally reload the page, but this interpretation might change if the current web page contains a button labeled "Refresh". This raises some important questions for future study.

First, how easy is it to choose weights and heuristics for a new application domain? It is likely to be impossible to tune the weights such that *every* user's expectations are *always* met. Nevertheless, we found that many weights used in the web prototype worked in the Word prototype as well, without change, which suggests that these weights are relatively domain independent. Similarly, although some heuristics are specific to certain basic types, like strings, it is also true that these types are common to many domains. We are also exploring learning weights automatically from a corpus of keyword commands and their desired interpretations.

Second, given that keyword commands may have different interpretations in different contexts, can users trust the system to interpret them without supervision? Can a user re-execute a script of commands that they wrote earlier, or that someone else gave to them, and expect the script to work the same way? The answer is generally yes, in our experience, but the risk of misinterpretation is not zero. It remains to be seen whether this risk is greater than other causes of bugs in user programs, such as encountering unexpected data or depending on a function whose behavior changes from one platform or version to another. Nevertheless, keyword commands may pose a tradeoff between ease of use and reliability, which has been a recent research concern in end-user programming [5]. One solution might be a compiler that converts a keyword command script into a lower-level representation, so that it can be run unsupervised without fear that its interpretation will vary.

Finally, how well does this approach scale to larger domains, with potentially more ambiguity? Human communication has many strategies for resolving ambiguity, including dialogue, shared context, or simply becoming more explicit and verbose. Many of these strategies will also be useful for disambiguating keyword commands. The critical technical challenge for scaling is probably the speed of parsing, which is an area we're actively working on.

USER STUDY

We believed that our command language was intuitive, and could be used without instructions, provided that the user was familiar with the domain. We therefore chose the web domain (which many end-users are familiar with) and implemented a command interface to test how well users could use the system to automate common web browsing tasks.

Participants

Our study involved 9 users, solicited from a public mailing list at a college campus. Seven were between 20 and 30 years old, while the other two were 49 and 56. We had three females and six males. Five were students (4 of these were computer science majors). The other four subjects had a range of occupations. All subjects were compensated for their time.

The subjects were also all experienced web users, and could type reasonably well. Almost every subject claimed to use the web almost every day (except one, who claimed to use the web a few times a week). Every subject had also been to the majority of the web sites involved in the study. Finally, each subject used typing-centered programs (like Word or an Instant Messenger) almost every day (again except for one, but even this user felt reasonably comfortable typing).

Programming experience amongst the users was divided into two groups. Two users had never written a program, and two had only written a program for a class. Each of the remaining five users had written multiple programs on their own, and was familiar with a number of different programming languages. We shall refer the first four users as *non-programmers*, and the last five users as *programmers*.

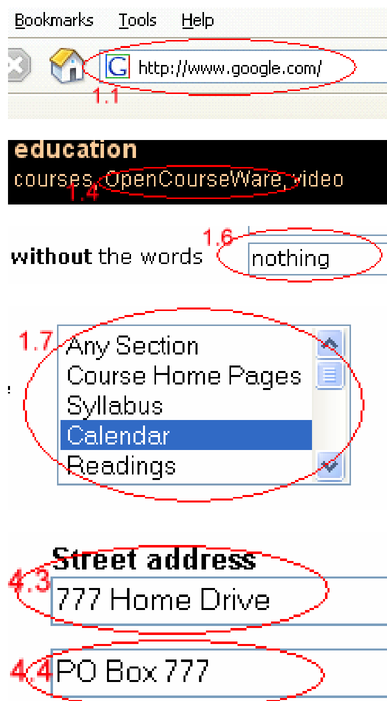

Figure 4: Examples of tasks in the user study. The user had to write a keyword command that would click or set the control circled in red. The red number is the task number.

Setup

Each subject sat at a computer loaded with the web domain prototype. We then handed them a set of instructions to read and tasks to complete.

Instructions: The instructions indicated that they should use only the command box (Figure 2a) to do each task, and not click or type directly into the web page. We also indicated that it was up to them to decide what to type into the command box. We did not offer any suggestions about what to type (except for two users, see *Modifications to Study* below).

Tasks: Each of the 36 tasks consisted of a red circle drawn on a screen shot of the web browser, indicating what to do (see Figure 4). For instance, if we wanted the user to navigate to a URL, we would circle a location bar loaded with that URL. We did this in order to minimize external hints about how the user should communicate with the system.

Even still, the circled text itself acted as a hint for many tasks, especially for following links and clicking buttons. In these cases, entering the text on the link or button itself was sufficient to click it. However, we did not tell the subjects that this was the case, and in fact, users often provided unnecessary words for these tasks (like the word "click").

We also included some more difficult tasks that required words not present in the red circle to complete. However, every task could be completed with a single keyword command.

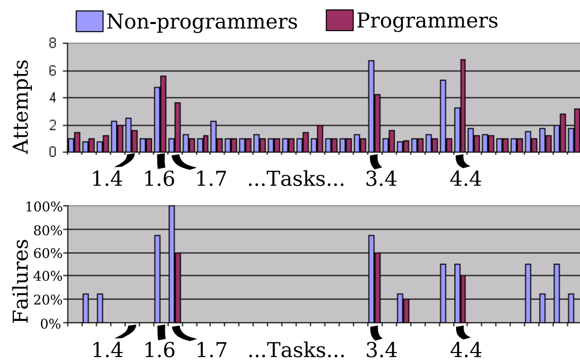

Figure 5: Average number of attempts made to accomplish each task (top chart), and average failure rate (bottom chart).

Modifications to Study: After running 6 users through the study, we noticed that people didn't use many verbs. For instance, subjects tended to enter text into form fields with expressions like ***without "nothing"*** rather than ***enter "nothing" into the without textbox***.

We wondered what effect it would have if we introduced a single initial suggestion of a command involving a verb. We decided to provide a hint for a task that every user had succeeded at (one of the easier tasks). We felt that this wouldn't overly contaminate the study, while giving us a small hint to satisfy our curiosity.

We therefore suggested to 2 users that they do the first task with ***go to google.com***. One user responded with "who needs verbs?", and proceeded to do this task with just ***google.com***. The other user took the suggestion, and used a verb in 10 tasks that no other user used a verb for, which may be cause to investigate this further.

Results

We recorded the number of attempts each user made to accomplish each task. We also recorded whether they ultimately succeeded, or eventually gave up. Note that we did not give users a time limit to accomplish tasks, but if they seemed at a loss for what to do, then we told them that they could skip the task and move on to the next one. If they did so, we counted this as giving up.

The non-programmer group succeeded at 84% of the tasks, and the programmer group succeeded at 95% of the tasks. (We found this difference statistically significant using a two-tailed t-test, with $p = 0.04$.) Each group averaged 1.7 attempts per task. Non-programmers completed 72% of the tasks on the first try, with only one command. The programmers achieved this for 77% of the tasks. If the system understood only JavaScript, and we had offered no instructions, we would have expected a completion rate around 0% for both groups.

Figure 5 shows the average number of attempts made to accomplish each task (top chart), coupled with the average failure rate for each task (bottom chart). The tasks with labels are referenced in the *Discussion* below.

Discussion

Certain tasks exposed important issues with the system.

Task 1.4: Everyone succeeded at this task, but it exposed an important problem with the tokenizer. The task was to click the "OpenCourseWare" link on MIT's homepage, which the tokenizer split into 3 tokens. However, this version did not add "OpenCourseWare" to the spelling dictionary. Hence, when users would type ***opencourseware***, the system would not split the token into ***open course ware***, and it would not match the link. In the new system, typing ***opencourseware*** is corrected by the spell checker into ***OpenCourseWare***, and the expression is then tokenized correctly.

Task 1.6: This is the first task with multiple failures. The task asked the user to enter the word "nothing" into a textbox labeled "without the words." Only 3 users completed this task on their first attempt.

All of the remaining subjects expected the system to have a notion of an input focus. They began the task by trying to focus the computer's attention on the appropriate textbox with commands like ***go to without the words***. We believe such a paradigm could be made to work within the bounds of the command language by adding commands like ***focusOn***, ***focusPrevious***, and ***focusNext*** (along with appropriate synonyms, e.g., ***go***). Even still, 3 of these users eventually succeeded at the task.

Task 1.7: This was really the second part of a two-part task. The task asked the user to select "Calendar" from a listbox. After doing so, they were meant to notice that the "Any Section" option was still selected in the web page, but not selected in the task illustration. At this point, they were meant to deselect the "Any Section" option.

Three subjects did not make this attempt, presumably because they thought it was a bug in the instructions or the webpage, or they did not notice it. One subject tried to access the listbox by focusing the computer's attention on it, which, as discussed previously, merits the addition of ***focus*** functions. The final three subjects used keywords which had not been added as synonyms for the proper functions in the system. The lesson learned here is that more synonyms need to be added, but we are encouraged to find that we did not need to add new functions in this case.

Task 3.4: This task seemed simple: it asked the user to enter a password into a password field. However, it exposed a bug in our choice of weights for the function scoring system. A brief description of this bug should prove instructive. Consider the input ***Password bloppy***. Five users tried this input, and it should have worked. Instead, the system clicked a link with the word "password" in it. Both of these interpretations needed to introduce an unnamed command, but the ***enter*** command cost 0.2, whereas the ***click*** command cost only 0.1. Of course, the click interpretation had to treat the word ***bloppy*** as an extraneous word, which cost another 0.1, whereas the ***enter*** interpretation treated ***bloppy*** as a string, costing 0. Both interpretations explained the word ***password*** as a keyword-list identifying a textbox or link.

The resulting score for ***click(findLink("password"))*** was 0.8, and the score for ***enter("bloppy", findTextbox("password"))*** was also 0.8. However, the ***click*** interpretation happened to appear first in the list, and the system went ahead with this interpretation, since the prototype afforded no means of disambiguation after the command was entered. We patched this problem by increasing the cost of extraneous words to 0.3, but ultimately we plan to implement the disambiguation dropdown we discussed.

Task 4.4: This was probably the hardest task in the study. It required the user to enter the address "PO Box 777" into the second street address field, shown at the bottom of Figure 4. People had difficulty identifying this field since it had no label of its own. Five subjects eventually succeeded, but 4 users did not. It is instructive to examine the potential reasons for these failures.

One user tried the approach of first focusing the computer's attention, and then typing. This user entered ***street address 2*** and then ***PO Box 777***. It is worth noting that if these commands had been entered as the single expression ***street address 2 PO Box 777***, they would have worked. It would also have worked if we had the ***focusOn*** command, along with a version of the ***enter*** command that accepts only a string.

Another user entered ***777 Home Drive*** into the first textbox successfully, and then issued the command ***next***. This attempt would also have been helped with the addition of ***focus*** commands, specifically ***focusNext***.

This same user also tried what turned out to be a common approach, which was to try entering both lines with a single command: ***777 Home Drive, PO Box 777 Street address***. In fact, 6 users made attempts of this sort. This is a harder problem to solve. One solution might consist of an ***enter*** command with 2 string arguments. However, we should note that all the users who attempted this approach eventually succeeded, except for two. One would have figured out a way if the system obeyed ***focus*** commands as discussed before. The other would have succeeded if not for a bug in the parser. To understand this bug, it is helpful to know that the most common type of expression that worked was: ***Street address 2 "PO Box 777"*** (3 people succeeded with commands of this sort). Now the expression of the user in question was: ***Street2=PO Box 777***. The problem is that this version of the tokenizer treated ***Street2*** as a single token. If the user had put a space between ***Street*** and ***2***, this command would have worked. The current prototype now supports this command by treating alpha and numeric sequences as separate tokens.

FUTURE WORK

We plan to continue to develop this technique in several directions. First, we believe there is a lot of room for improvement to the algorithm, now that we've convinced ourselves that the algorithm is useful. Also, we want to add new features to the algorithm. In particular, we would like to support variable declarations which are referenced in the same command. This may allow for for-loop constructs.

Another goal is to integrate each prototype into its target domain, and test usability. In Chickenfoot, we would like to see if end-users without JavaScript experience can create simple

scripts. In Word, we would like to test how keyword commands compare with navigating a large menu system.

We also believe the technique may be useful for programming in Java; consider typing ***time***, and getting back ***System.-currentTimeMillis()***. We believe such translations may be possible, and useful.

Finally, we would like to explore the potential of using the translation system as a generic backend for a speech recognition system. We believe that the general framework may make it easy to expose core sets of functionality of various applications to end-users.

CONCLUSION

We have scratched the surface of a domain with great potential: translating keyword commands into executable code. We have described an algorithm for performing such translations in a reasonable time for small applications, and we have demonstrated that users can form commands that are interpretable by the system, without any training.

ACKNOWLEDGMENTS

We would like to thank all the participants in our user study. We also appreciate all the helpful suggestions we have received from the anonymous referees, members of the UID group, and other friends and peers. This work was supported in part by the National Science Foundation under award number IIS-0447800. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. Apple Computer. *Automator.* http://www.apple.com/-macosx/features/automator/, accessed June 28, 2006.
2. Ballard, B., and Biemann, A. Programming in Natural Language: NLC as a Prototype. *ACM/CSC-ER Annual Conference*, 228-237. 1979.
3. Bruckman, A., Community Support for Constructionist Learning. *Computer Supported Cooperative Work*, 7(1-2), 47–86, Jan. 1998.
4. Bruckman, A., Edwards, E. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. *CHI '99*, pp. 207–214.
5. Burnett, M., Cook, C., and Rothermel, G. End-User Software Engineering. *Commun. ACM*, 47(9), 53–58, Sept. 2004.
6. Bolin, M., Webber, M., Rha, P., Wilson, T., Miller, R. Automation and customization of rendered web pages. *UIST 2005*, pp. 163–172.
7. Cypher, A., Ed. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.
8. Erwig, M., Abraham, R., Cooperstein, I., and Kollmansberger, S. Automatic generation and maintenance of correct spreadsheets. *ICSE 2005*, pp. 136–145.
9. Katz, B., Felshin, S., Yuret, D., Ibrahim, A., Lin, J., Marton, G., McFarland, A., and Temelkuran, B. Omnibase: Uniform Access to Heterogeneous Data for Question Answering. *NLDB 2002*, pp. 230-234.
10. Pausch, R., et al. Alice: A Rapid Prototyping System for 3D Graphics. *IEEE Computer Graphics and Applications*, 15(3), 8–11, May 1995.
11. Kelleher, C. and Pausch, R., Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2), 83–137, 2005.
12. Ko, A.J. and Myers, B.A. Designing the Whyline: A Debugging Interface for Asking Why and Why Not Questions. *CHI 2004*, pp. 151–158.
13. Ko, A.J., Myers, B.A., and Aung, H. Six Learning Barriers in End-User Programming Systems. *VL/HCC 2004*, pp. 199–206.
14. Lieberman, H., Ed. *Your Wish is My Command: Programming By Example.* Morgan Kaufmann, San Francisco, CA, 2001.
15. Liu, H., and Lieberman, H., Programmatic Semantics for Natural Language Interfaces. *CHI 2005*, pp. 1597–1600.
16. Miller, L., Natural Language Programming: Styles, Strategies, and Contrasts. *IBM Systems Journal*, 1981.
17. Miller, P., Pane, J., Meter, G., and Vorthmann, S. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments*, 4(2), 140–158, 1994.
18. Mandelin, D., Xu, L., Bodik, R., and Kimelman, D. Jungloid Mining: Helping to Navigate the API Jungle. *PLDI '05*, pp. 48–61.
19. Myers, B., Pane, J., and Ko, A., Natural Programming Languages and Environments. *CACM*, 47(9), 47–52, Sept. 2004.
20. Phalgune, A., Kissinger, C., Burnett, M., Cook, C., Beckwith, L., and Ruthruff, J.R. Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User Programmers. *VL/HCC 2005*, pp. 45–52.
21. Porter, M., An algorithm for suffix stripping, *Program*, 14(3), pp 130-137, 1980.
22. Price, D., Riloff E., Zachary J., and Harvey B. NaturalJava: A Natural Language Interface for Programming in Java. *IUI 2000*, pp. 207–211.
23. Sammet, J., The Use of English as a Programming Language. *CACM*, 9(3), 228-230. 1966.
24. Teitelbaum, T. and Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. *CACM*, 24(9), 563–573, 1981.
25. Wiedenbeck, S., Engebretson, A. Comprehension strategies of end-user programmers in an event-driven application. *VL/HCC 2004*, pp. 207–214.