



MIT Open Access Articles

Sloppy programming

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation	Little, Greg, et al. "Sloppy Programming." No Code Required, Elsevier, 2010, pp. 289–307.
As Published	http://dx.doi.org/10.1016/b978-0-12-381541-5.00015-8
Publisher	Elsevier
Version	Author's final manuscript
Citable link	https://hdl.handle.net/1721.1/121718
Terms of Use	Creative Commons Attribution-NonCommercial-NoDerivs License
Detailed Terms	http://creativecommons.org/licenses/by-nc-nd/4.0/

Sloppy Programming

Greg Little¹, Robert C. Miller¹, Victoria N. Chou¹, Michael Bernstein¹, Allen Cypher², Tessa Lau²

¹MIT Computer Science and Artificial Intelligence Laboratory
²IBM Almaden Research Center

Introduction

When a user enters a query into a web search engine, they do not expect it to return a syntax error. Imagine a user searching for "End User Programing" and getting an error like: Unexpected token "Programing". Not only do users not expect to see such an error, but they expect the search engine to suggest the proper spelling of "Programing".

In effect, users expect the search engine to try everything within its power to make sense of their input. The burden is on the search engine to make the user right. People have come to expect this behavior from search engines, but they do not expect this behavior from program compilers or interpreters.

When a novice programmer enters "print hello world" into a modern scripting language, and the computer responds with "SyntaxError: invalid syntax", the attitude of many programmers is that the user did something wrong, rather than that the computer did something wrong. In this case, the user may have forgotten to put quotes and parenthesis around "hello world", depending on the underlying formal language. Programmers do not often think that the computer forgot to search for a syntactically correct expression that most closely resembled "print hello world".

This attitude may make sense when thinking about code that the computer will run without supervision. In these cases, it is important for the programmer to know in advance exactly how each statement will be interpreted.

However, programming is also a way for people to communicate with computers on a day-to-day basis, in the form of scripting interfaces to applications. In these cases, commands are typically executed under heavy supervision. Hence, it is less important for the programmer to know in advance precisely how a command will be interpreted, since they will see the results immediately, and they can make corrections. Unfortunately, scripting interfaces and command prompts typically use formal languages, requiring users to cope with rigid and seemingly arbitrary syntax rules for forming expressions. One 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.

Not only do scripting interfaces use formal languages, but different applications often use different formal languages -- Python, Ruby, Javascript, sh, csh, Visual Basic, ActionScript, to name a few. These languages are often similar—many are based on C syntax— but they are different enough to cause problems.

In addition to learning the language, users must also learn the Application Programmer Interface (API) for the application they want to script. This can be challenging, since APIs

are often quite large, and it can be difficult to isolate the portion of the API relevant to the current task.

We propose that instead of returning a syntax error, an interpreter should act more like a web search engine. It should first search for a syntactically valid expression over the scripting language and API. Then it should present this result to the user for inspection, or simply execute it, if the user is feeling lucky. We call this approach *sloppy programming*, a term coined by Tessa Lau at IBM.

This chapter continues with an explanation of sloppy programming. We then present several prototype systems which use the sloppy programming paradigm, and discuss what we learned from them. Before concluding, we present a high level description of some of the algorithms that make sloppy programming possible, along with some of their tradeoffs in different domains.

Sloppy Programming

The essence of sloppy programming is that the user should be able to enter something simple and natural, like a few keywords, and the computer should try everything within its power to interpret and make sense of this input. The question then becomes: how can this be implemented? One key insight is that many things people would like to do, when expressed in English, resemble the programmatic equivalents of those desires. For example, consider the case where a user is trying to write a script to run in the context of a webpage. They might use a tool like Chickenfoot [27] which allows end-users to write web customization scripts. Now let's say the user types "now click the search button". This utterance shares certain features with the Chickenfoot command `click("search button")`, which has the effect of clicking a button on the current webpage with the label "search". In particular, a number of keywords are shared between the natural language expression, and the programming language expression, namely "click", "search", and "button".

We hypothesize that this keyword similarity may be used as a metric to search the space of syntactically valid expressions for a good match to an input expression, and that the best matching expression is likely to achieve the desired effect. This hypothesis is the essence of our approach to implementing sloppy programming. One reason that it may be true is that language designers, and the designers of APIs, often name functions so that they read like English. This makes the functions easier to find when searching through the documentation, and easier for programmers to remember.

Of course, there are many syntactically valid expressions in any particular programming context, and so it is not easy to exhaustively compare each one to the input. However, we will show later some techniques that make it possible to approximate this search to varying degrees, and that the results are often useful.

Interface

Our prototype systems have used two main interfaces for sloppy programming. The first is a command line interface, where the user types keywords into a command box and the system interprets and executes them immediately. Note that although commands traditionally require a verb, the sloppy queries may be suggestive, without using an imperative verb. For example, consider the query "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 sloppy command line system would try to perform this translation automatically, and then execute the Visual Basic behind the scenes to achieve the user's intent.

The second interface is an extension to autocomplete, where the system replaces keywords in a text editor with syntactically correct code. This interface is aimed more at expert programmers using an integrated development environment (IDE). As an example of this interface, imagine that a programmer wants to add the next line of text from a stream to a list, using Java. A sloppy completion interface would let them enter "add line" and the system might suggest something like "lines.add(in.readLine())".

Benefits

Sloppy programming seeks to address many of the issues raised previously. To illustrate, consider the "left margin 2 inches" example. First, note that we don't require the user to worry about strict requirements for punctuation and grammar in their expression. They should be able to say something more verbose, like "set the left margin to 2 inches", or expressed in a different order, like "2 inches, margin left". Second, the user should not need to know the syntactic conventions for method invocation and assignment in Visual Basic. The same keywords should work regardless of the underlying scripting language. Finally, the user should not have to search through the API to find the exact name for the property they want to access. They also should not need to know that LeftMargin is a property of the PageSetup object, or that this needs to be accessed via the ActiveDocument.

Another advantage of sloppy programming is that it accommodates pure text. The benefits of pure text are highlighted in [28], and 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.

Another benefit of using a textual representation is that it allows lists of sloppy queries (a sloppy program) 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. Sloppy programs might offer the same ability for arbitrary applications—you could store or share a sloppy program 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 queries 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).

Sloppy programming also offers benefits to expert programmers as an extension to autocomplete, by decreasing the cognitive load of coding in several ways. First, sloppy queries are shorter than code, and easier to type. Second, the user does not need to recall all the lexical components (e.g. variable names and methods) involved in an expression,

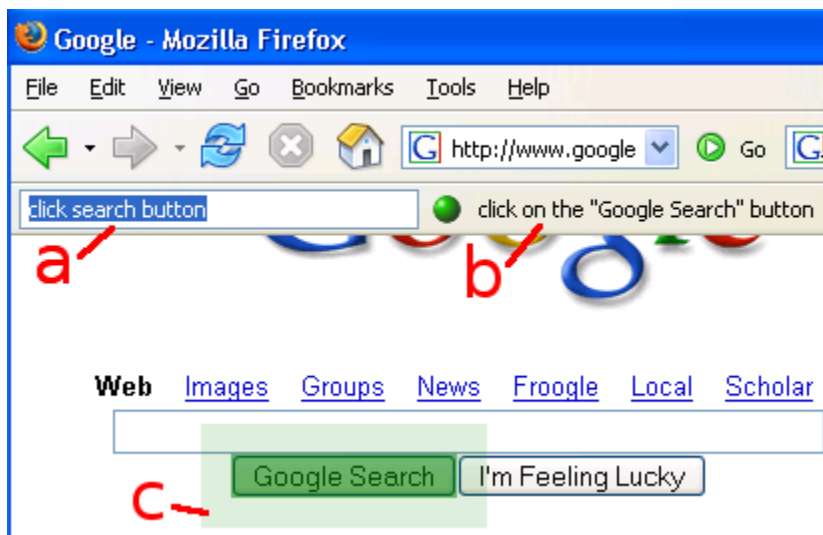
since the computer may be able to infer some of them. Third, the user does not need to type the syntax for calling methods. In fact, the user does even need to know the syntax, which may be useful for users who switch between languages often.

Systems

Over the past few years, we have built a number of systems to test and explore the capabilities of sloppy programming. Three of these systems take the form of command line interfaces for interacting with websites. These systems explore usability aspects of sloppy programming without worrying too much about the implementation, since the underlying command set is fairly small, so even a naive implementation is fast enough for real-time use. The fourth prototype explores sloppy programming in an integrated development environment for Java, where the system needs to generate valid Java expressions in real world Java projects. This system required thinking more deeply about the algorithm, though we'll mainly discuss the UI challenges relevant to the IDE domain here.

Sloppy Web Command Line

Our first prototype system [31] tested the hypothesis that a sloppy programming command line interface was intuitive, and could be used without instructions, provided that the user was familiar with the domain. The web domain was chosen because many end-users are familiar with it. The system takes the form of a command interface which users can use to perform common web browsing tasks. The system is a plugin for the Firefox web browser. Important UI elements are shown here:



The GUI for this prototype consists of a textbox affording input (a), and an adjacent horizontal bar allocated for textual feedback (b). The system also generates an animated acknowledgment in the web page around the html object affected by a command as shown surrounding the "Google Search" button in the figure above (c).

The functions in the web prototype map to commands in Chickenfoot [27]. At the time we

built this prototype, there were 18 basic commands in Chickenfoot, including `click`, `enter` and `pick`. The `click` command takes an argument describing a button on the webpage to click. The `enter` command takes two arguments, one describing a textbox on the webpage, and another specifying what text should be entered there. The `pick` command is used for selecting options from HTML comboboxes.

In order to cover more of the space of possible sloppy queries that the user could enter, we included synonyms for the Chickenfoot command names in the sloppy interpreter. For instance, in addition to the actual Chickenfoot function names like "enter", "pick" and "click", we added synonyms for these names like "type", "choose" and "press", respectively.

For the most part, users of this prototype were able to use the system without any instructions, and accomplish a number of tasks. The following is an image taken directly from the instructions we handed to users during a user study:

The image shows a search interface with several sections. The 'Find results' section has four radio button options: 'related to all of the words', 'related to the exact phrase', 'related to any of the words', and 'not related to the words'. The first option is selected, and the text 'lisa simpson' is entered in the adjacent text box. The 'Size' section has a dropdown menu set to 'large'. The 'Filetypes' section has a dropdown menu set to 'GIF files'. The 'Coloration' section has a dropdown menu set to 'grayscale'. The 'Domain' section has an empty text box. The 'SafeSearch' section has three radio button options: 'No filtering' (selected), 'Use moderate filtering', and 'Use strict filtering'. Red circles are drawn around the text box containing 'lisa simpson', the 'large' dropdown, the 'GIF files' dropdown, the 'grayscale' dropdown, and the 'No filtering' radio button.

The red circles indicates changes the user is meant to make to the interface (We tried to tell them what to do with pictures rather than words so as not to bias their choice of words). Here is a sequence of sloppy instructions for performing these tasks that were generated by different users in the study, that the system interpreted correctly: "field lisa simpson", "size large", "Return only image files formatted as GIF", "coloration grayscale", "safesearch no filtering".

However, we did discover some limitations. One limitation is that the users were not sure at what level to issue commands. If they wanted to fill out the form above, should they issue a high level command to fill out the entire form, or should they start at a much lower level, first instructing the system to shift its focus to the first textbox. In our study, users never tried issuing high levels commands (mainly because they were told that each red circle represented a single task). However, some users did try issuing commands at a lower level, first trying to focus the systems attention on a textbox, and then entering text there. In this case, we could have expanded the API to support these commands, but in general, it is difficult to anticipate at what level users will expect to operate in a system like this.

Another limitation is that users were not always able to think of a good way of describing certain elements within an HTML page, like a textbox with no label. Consider for example the second task shown here:

Street address
777 Home Drive

PO Box 777

City

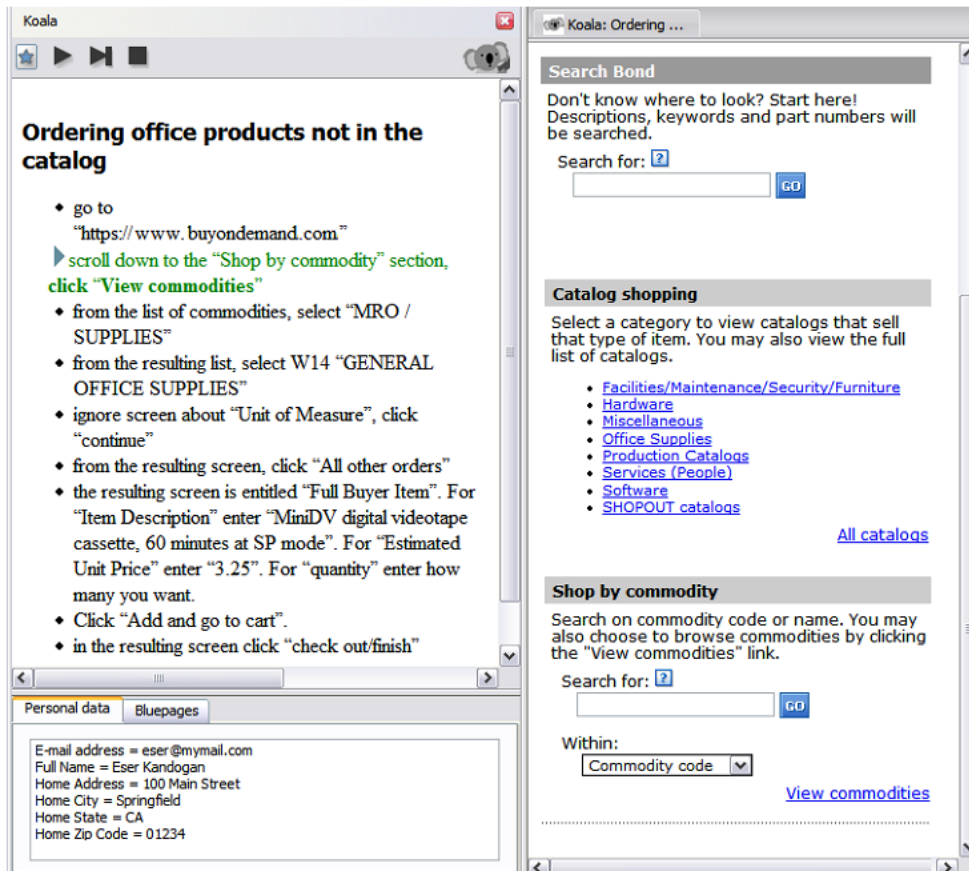
State / Province
Arizona

Zip / Postal code

We also found that despite our best efforts to support the wide range of ways that people could express things, we still found people trying things we didn't expect. Fortunately, in many cases, they would try again, and soon land upon something that the computer understood. This sort of random guessing rarely works with a formal programming language, and is one promising aspect of sloppy programming.

Koala

Koala [29] (eventually released as CoScripter [30]) was built to explore the synergy of combining several programming paradigms, of which sloppy programming was one. The other paradigms include programming by demonstration, and the use of a wiki as a form of end user source control. Koala is an extension to Firefox. It appears as a sidebar to the left of the current webpage as shown here:



The original Koala system experimented with additional user interface techniques to assist users with the sloppy programming paradigm. Because Koala's interpreter is sometimes wrong, it was also important to implement several techniques to help the user know what the interpreter was doing and allow the user to make corrections. These techniques are illustrated in the following example from the figure above. Suppose the user has selected the highlighted line: scroll down to the "Shop by commodity" section, click "View commodities". Koala interprets this line as an instruction to click a link labeled "View commodities". At this point, the system needs to make two things clear to the user: what is the interpreter about to do, and why?

The system shows the user what it intends to do by placing a transparent green rectangle around the View commodities link, which is also scrolled into view. The system explains why by highlighting words in the keyword query that caused this link to be chosen. In this case, the words click, view and commodities were associated with the link, so the system makes these words bold: scroll down to the "Shop by commodity" section, **click "View commodities"**.

If the interpretation was wrong, the user can click the triangle to the left of the line, which expands a list of alternate interpretations. These interpretations are relatively unambiguous instructions generated by the interpreter:

- click the "View commodities" link

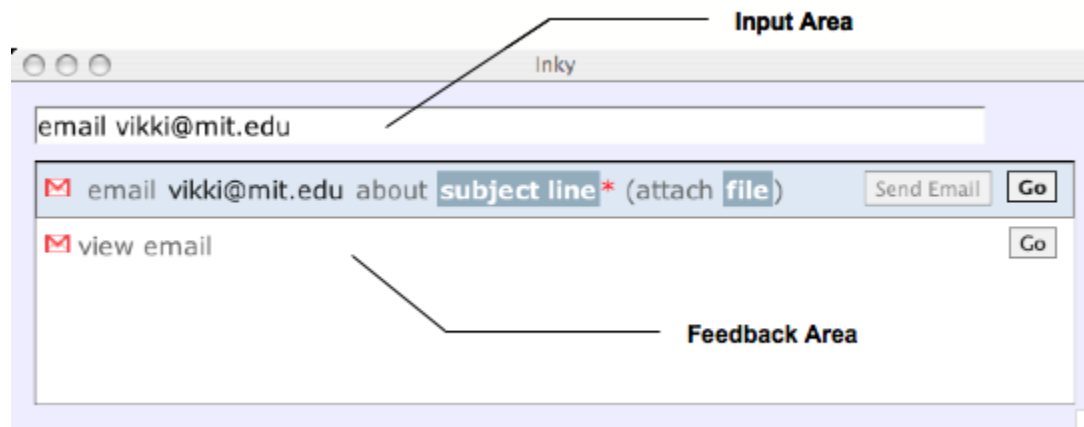
- click the "View contracts" link
- click the "Skip to navigation" link

When the user clicks on any of these lines, the system places a green rectangle over the corresponding HTML control. If the line is the correct interpretation, the user can click the Run or Step button to execute it. If not, they may need to edit the line. Failing that, they can add the keyword you (e.g., you click the "View commodities" link) which the interpreter uses as a cue to leave execution to the user.

Note that Koala also included a feature to allow users to record commands by interacting with webpages, rather than writing any sloppy code. However, the system would sometimes record a command as a sloppy command that the sloppy interpreter would then fail to execute correctly. This turned out to be a problem, and motivated CoScripter's current use of templates to match user commands.

Inky

Inky takes the idea of a web command interface to a new level, seeking to explore a sloppy command interface for higher level commands. Inky takes the form of a dialog that pops up when the user presses Control-Space in the web browser. This keyboard shortcut was chosen because it is generally under the user's fingers, and because it is similar to the Quicksilver shortcut (Command- Space on the Mac).



The Inky window has two areas: a text field for the user to type a command, and a feedback area that displays the interpretations of that command. The Inky window can be dismissed without invoking the command by pressing Escape or clicking elsewhere in the browser.

Inky Commands

A command consists of keywords matching a web site function, along with keywords describing its parameters. For example, in the command "reserve D463 3pm", the reserve keyword indicates that the user wants to make a conference room reservation, and D463 and 3pm are arguments to that function.

Like the other sloppy interpreters discussed previous, the order of keywords and parameters

is usually unimportant. The order of the entered command matters only when two or more arguments could consume the same keywords. For example, in the command "reserve D463 3pm 1 2 2007" it is unclear if 1 is the month and 2 is the date or vice versa. In "find flights SFO LAX", it is unclear which airport is the origin and which is the destination. In these cases, the system will give higher rank to the interpretation that assigns keywords to arguments in left-to-right order, but other orderings are still offered as alternatives to the user. Commands can use synonyms for both function keywords and arguments. For example, to "reserve D463 at 3pm", the user could have typed "make reservation" instead of "reserve", used a full room number like 32-D463 or a nickname like "star room", and used various ways to specify the time, such as 15:00 and 3:00.

Function keywords may also be omitted entirely. Even without function keywords, the arguments alone may be sufficient to identify the correct function. For example, D463 15:00 is a strong match for the room-reservation function because no other function takes both a conference room and a time as arguments.

The Inky prototype includes 30 functions for 25 web sites, including scheduling (room reservation, calendar management, flight searches), email (reading and sending), lookups (people, word definitions, Java classes), and general search (in search engines and ecommerce sites). Most of the functions included in the prototype are for popular web sites; others are specific to web interfaces (like a room reservation website) at the lab of the developer. Argument types specifically detected by Inky include dates, times, email addresses, cities, states, zip codes, URLs, filenames, and room names. Examples of valid commands include "email vikki@mit.edu Meeting Right Now!" (to send email), "java MouseAdapter" (to look up Java API documentation), "define fastidious" (to search a dictionary), "calendar 5pm meeting with rob" (to make a calendar event), and "weather cambridge ma" (to look up a weather forecast).

Inky Feedback

As the user types a command, Inky continuously displays a ranked list of up to five possible interpretations of the command (Figure 2). Each interpretation is displayed as a concise, textual sentence, showing the function's name, the arguments the user has already provided, and arguments that are left to be filled in. The interpretations are updated as the user types in order to give continuous feedback, as shown here:

```
> email
M view email Go
M email email address* about subject line* (attach file) Send Email Go

> email vikki@mit.edu
M email vikki@mit.edu about subject line* (attach file) Send Email Go

> email vikki@mit.edu movie at 3pm today
M email vikki@mit.edu about "movie at 3pm today" (attach file) Send Email Go
```

The visual cues of the interpretation were designed to make it easier to scan. A small icon indicates the website that the function automates, using the favicon image displayed in the

browser address bar when that site is visited. Arguments already provided in the command are rendered in black text. These arguments are usually exact copies of what the user typed, but may also be a standardized version of the user's entry in order to clarify how the system interpreted it. For example, when the user enters "reserve star room", the interpretation displays "reserve D463" instead to show that the system translated star room into a room number.

Arguments that remain to be specified appear as white text in a dark box. Missing arguments are named by a word or short phrase that describes both the type and role of the missing argument. If a missing argument has a default value, a description of the default value is displayed, and the box is less saturated. In the figure below, "name, email, office, etc." is a missing argument with no default, while "this month" is an argument which defaults to the current month.



reserve **room*** on **this month** **this date, this year** **at this time** (repeats **never**) for **description***
directory lookup **name, email, office, etc.**

For functions with persistent side effects, as opposed to merely retrieving information, Inky makes a distinction between arguments that are required to invoke the side effect and those that are not. Missing required arguments are marked with a red asterisk, following the convention used in many web site forms. In the figure above, the room and description are required arguments. Note that the user can run partial commands, even if required arguments are omitted. The required arguments are only needed for running a command in the mode that invokes the side effect immediately. The feedback also distinguishes optional or rarely-used arguments by surrounding them by parentheses, like (repeats never) argument in the figure above. It should be noted that this feedback does not dictate syntax. The user does not need to type parentheses around these arguments. If the user did type them, however, the command could still be interpreted, and the parentheses would simply be ignored.

Quack Eclipse Plugin

Our fourth and final example of a sloppy programming system is a departure from the previous systems, because it targets both a different domain (Java programming rather than web browsing) and a different user audience (professional programmers rather than end users). Quack [26] is a plugin for the Eclipse integrated development environment that explores both the speed and usability of sloppy programming in the domain of general purpose Java programming. The plugin integrates with Eclipse's Java editor, and provides a new kind of autocomplete feature.

With the Quack plugin installed, the user may enter sloppy queries directly into their source code. The following figure shows the user entering the sloppy query "add line":

```

public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        add line
    }
    return array;
}

```

Next, the user presses Ctrl-Space to bring up Eclipse's autocomplete menu, which also triggers the Quack plugin. It then does the following:

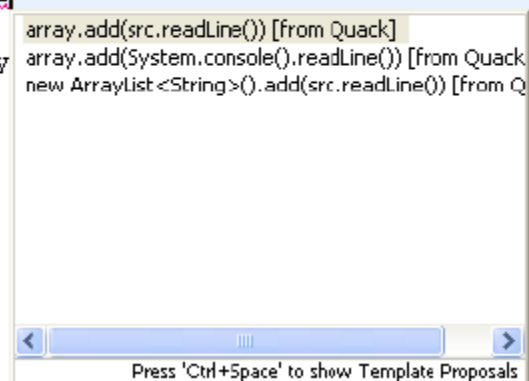
1. Quack updates its internal model of the current context with local variables visible at the cursor location, as well as classes named in the current source file.
2. Quack figures out where the keyword query begins. (The end of the query is assumed to be the current cursor position.) We have experimented with various heuristics for finding the start of the query, including the left-most Java compilation error on the current line, which occurs on the keyword `add` in this example. Another heuristic considers the text on the current line preceding the cursor, and finds the longest suffix that contains only tokens that look like keywords (namely words and numbers, but not punctuation).
3. Quack determines what Java types are valid for an expression at this location. In this example, the keywords are not nested in a subexpression, so any return type is valid. If the user had instead typed the query "in ready" into the condition for the while loop, then the system would expect a boolean return type, because the condition for a while loop must be a boolean.

The system then plugs this information into an algorithm that suggests valid Java expressions matching the keywords. These expressions then appear in Eclipse's autocomplete menu for the user to choose from. In this example, the top-most suggestion is "lines.add(in.readLine())".

```

public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        add line
    }
    return array;
}

```



Once the user selects this option, Quack replaces the sloppy command with the selected code as shown:

```
public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        array.add(src.readLine());
    }
    return array;
}
```

If the user is not satisfied with the result, Ctrl-Z can undo it.

Users of the Quack plugin found it useful in a number of ways. First, it allowed a way of recalling a function name given other constraints, like input parameters. It also acted as a shortcut for small common expressions, like `System.out.println("hello")`, which users could write by typing `"out 'hello'"`, and then invoking Quack. Some users also found it nice to type commands without worrying about capitalization or punctuation, so they might type `"vector element at j"` and have this completed into `"fIVector.elementAt(j)"`.

Although we had wanted Quack to help people using unfamiliar APIs, our studies suggest the current implementation did not work well for this. Part of the reason stems from users not knowing which synonyms to use for functions like `"add"` or `"append"`, but more confusion stems from simply not understanding how the API works at a deeper level.

Another problem is that the current plugin does not offer a lot of feedback when things go wrong. If the expression that the user wants does not show up in the list, it is not always clear why the plugin failed to find it. Sometimes the expression the user is looking for isn't actually valid yet, possibly because some of the variables or functions involved haven't been defined or imported yet, or are private when they need to be public. It is certainly the goal for sloppy programming that the system work hard to address issues like this, but this is still future work at the time of this writing.

A third problem to think about is that it is not always easy for the user to know if the results coming back from Quack are correct. It is easier using standard autocomplete, since the user only needs to verify a single token, but Quack is suggesting an entire expression, and the temptation is high to assume the entire expression is correct without reading it over to make sure.

Algorithm

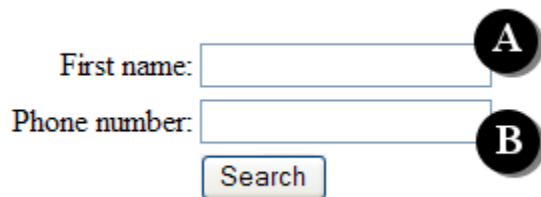
All of the systems described above need some way of converting a sloppy query into syntactically valid code. This section describes two approaches for performing these translations. The first approach assumes that the possible interpretations for a sloppy command is limited mainly by what is actually possible in the current context. This is the case in a typical form-based webpage, where the number of commands is roughly proportional to the number of form widgets. This algorithm was used in the Koala system.

The second approach assumes that the possible interpretations in the current context is extremely large, and that the only effective way of searching through it is to guide a search

using the keywords that appear in the sloppy query. For this approach, we focus on a slow but easy to understand version that was actually used in the Web Command prototype. We then provide high level insights into some improvements that can be made to the algorithm, along with references for more details about these improvements. The more advanced algorithm was used in the Inky and Quack prototypes.

Koala Algorithm

We describe this algorithm in three basic steps, using the example instruction "type Danny into first name field" on a simple web form shown here:



The diagram shows a web form with two text input fields and a search button. The first field is labeled "First name:" and is marked with a black circle containing the letter "A". The second field is labeled "Phone number:" and is marked with a black circle containing the letter "B". Below the second field is a button labeled "Search".

First, the interpreter enumerates all the possible actions associated with various HTML objects in the document, such as links, buttons, text boxes, combo-box options, check boxes, and radio buttons. For each of these objects, the interpreter associates a variety of keywords, including the object's label, synonyms for the object's type, and action verbs commonly associated with the object. For instance, the "First name" field of the web form shown above would be associated with the words "first" and "name", because they are part of the label. It would also associate "textbox" and "field" as synonyms of the field's type. Finally, the interpreter would include verbs commonly associated with entering text into a text field such as "enter", "type", and "write".

Next, the interpreter searches for the object that matches the greatest number of keywords with the sloppy input sequence. In the example, textbox A would match the keywords "type", "first", "name" and "field", with a score of 4. In contrast, textbox B would only match the keywords "type" and "field", with a score of 2.

Finally, the system may need to do some post-processing on the slop to extract arguments (e.g. "Danny"). The key idea of this process is to split the slop into two parts, such that one part is still a good match to the keywords associated with the web object, and the second part is a good match for a string parameter using various heuristics (e.g. has quotes around it, or is followed/preceded by a preposition). In this example, the parts are: "type into first name field" (with 4 keyword matches), and "Danny" (which gets points for being followed by the preposition "into" in the original slop).

This technique is surprisingly effective, since a web page provides a very small search space.

Web Command Line Algorithm

Functions are the building blocks in this algorithm. Each function returns a single value of a certain type, and accepts a fixed number of arguments of certain types. Literals, like strings or integers, are represented as functions that take no arguments and return an appropriate type. We can implement optional arguments for functions with function overloading.

Functions can have many names, usually synonyms of the true function name. For instance, the `enter` command in the web prototype has names like "type", "write", "insert", "set", and the `=` symbol, in addition to its true name "enter". Functions for literals may be thought of as functions with names corresponding to the textual representation of the literal. These names are matched programmatically with regular expressions, e.g., integers are matched with `[0-9]+`.

Arguments can also have names (and synonyms), 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 only into the parent function. This may be useful for functions that take two arguments of the same type, like the `copy` command in most operating systems. In this case, the idea would be to name one argument something like "from", and the other argument something like "to".

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` to form the tokens "Left" and "Margin". 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" and "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" and "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 a desired return type which is a supertype of all other types (unless the context of the sloppy commands restricts the return type).

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. One way to handle this is 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. We decided upon these values by hand in all of the systems built so far, since we do not have a corpus of sloppy commands to measure against. In the future, these values should probably be learned from a corpus of user generated sloppy inputs.

Algorithm Optimizations

The algorithm described above is too slow in practice for applications like the Quack plugin, which builds expressions over hundreds of object types and thousands of functions. In order to deal with the larger scale, we have explored a number of ways of making the algorithm more efficient, at the expense of making it less accurate.

The first step is to impose an arbitrary limit on the depth of expressions. Then we use a

dynamic program to calculate a subset of the best expression trees of this depth. We start by calculating the best expression trees of depth 0 (i.e. functions that return the required type, but take no parameters). We only keep a set number of functions that return each type. Then we find expressions of depth 1 by looking at functions that return each possible type, and filling in the parameters with good functions that we kept at depth 0. Again, we only keep a set number of functions for each type at depth 1. Then we repeat this process up to a chosen depth. The Quack plugin had a relatively shallow depth of 2.

After filling in the dynamic programming table, we use an off-the-shelf search algorithm to find the best expression tree. Quack used A* for this search. More details about the algorithm can be found in [26].

Related Work

Sloppy programming may be viewed as a form of natural programming (not to be confused with natural language programming, though there is some overlap). 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 nonprogrammers 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. Sloppy programming also relies on a knowledge base, but representing just the application domain, rather than global common sense.

The sloppy programming algorithms, each of which is essentially a search through the application's API, are 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 object to an Image object in Java). 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. Sloppy programming algorithms must also search an API to generate valid code, but the query is richer, since keywords from the query 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]. Sloppy programming does not address these issues, and may even force tradeoffs in some of them. In particular, sloppy

programs may not be as reliable.

Conclusion

The systems described in this chapter still just scratch the surface of a domain with great potential: translating sloppy commands into executable code. We have described potential benefits to end-users and expert programmers alike, as well as advocated a continued need for textual command interfaces. We have also described a number of prototypes exploring this technology, and discussed what we learned from them, including the fact that users can form commands for some of these systems without any training. Finally, we gave some high level technical details about how to go about actually implementing sloppy translation algorithms, with some references for future reading.

Acknowledgements

We thank all the members of the UID group at MIT and the User Interface group the IBM Almaden Research Center who provided valuable feedback on the ideas in this chapter and contributed to the development of the prototype systems, particularly Lydia Chilton, Max Goldman, Max Van Kleek, Chen-Hsiang Yu, James Lin, Eben M. Haber, Eser Kandogan. This work was supported in part by the National Science Foundation under award number IIS-0447800, and by Quanta Computer as part of the T-Party project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

Bibliography

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. Natural- Java: 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.

26. Little, G., and Miller, R.C., "Keyword Programming in Java." *Journal of ASE*, 2008.
27. Bolin, M., et al. "Automation and Customization of Rendered Web Pages." *UIST*, 2005, pp. 163-172.
28. Robert C. Miller. *Lightweight Structure in Text*. PhD thesis, Carnegie Mellon University, 2002.
29. Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. "Koala: Capture, Share, Automate, Personalize Business Processes on the Web." *CHI*, 2007
30. Leshed, G., Haber, E. M., Matthews, T., and Lau, T. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems (Florence, Italy, April 05 - 10, 2008)*. *CHI '08*.
31. Greg Little, and Robert C. Miller. "Translating Keyword Commands into Executable Code." *UIST*, 2006