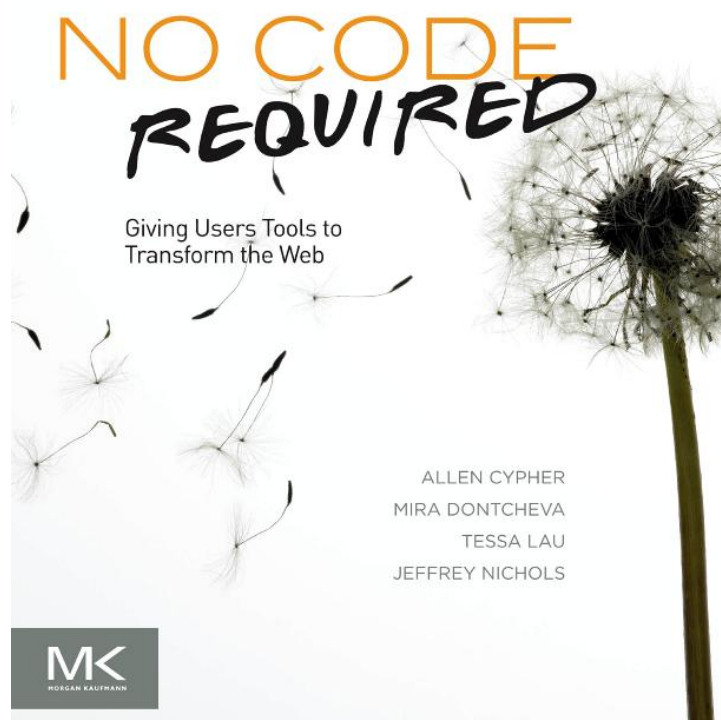


**Provided for non-commercial research and educational use only.
Not for reproduction, distribution or commercial use.**

This chapter was originally published in the book *No Code Required: Giving Users Tools to Transform the Web*, published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues who know you, and providing a copy to your institution's administrator.



All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

<http://www.elsevier.com/locate/permissionusematerial>

From: Greg Little¹, Robert C. Miller¹, Victoria H. Chou¹, Michael Bernstein¹, Tessa Lau², Allen Cypher², Sloppy programming. In: Allen Cypher, Mira Dontcheva, Tessa Lau and Jeffrey Nichols, editors: *No Code Required: Giving Users Tools to Transform the Web*. Burlington: Morgan Kaufmann, 2010, pp. 289-307.

ISBN: 978-0-12-381541-5

© Copyright 2010 Elsevier Inc.

Morgan Kaufmann.

Sloppy programming

15

Greg Little,¹ Robert C. Miller,¹ Victoria H. Chou,¹ Michael Bernstein,¹ Tessa Lau,² Allen Cypher²

¹MIT CSAIL

²IBM Research–Almaden

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 loosely grammatical commands of this sort “sloppy commands.” We discuss several prototypes that implement sloppy programming, translating sloppy commands directly into executable code. We also discuss the algorithms used in these prototypes, expose their limitations, and propose directions for future work.

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 would they not expect to see such an error, but they would expect the search engine to suggest the proper spelling of “Programing”. 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 novice did something wrong, rather than that the computer did something wrong. In this case, the novice may have forgotten to put quotes and parentheses 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 daily, 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 similarly inscrutable 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, and ActionScript, to name a few. These languages are often similar – many are based on C syntax – but they are different enough to cause problems. For instance, JavaScript assumes variables are global unless explicitly declared to be local with “var”, whereas Python assumes variables are local unless declared with “global”.

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 related work, followed by a deeper 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.

RELATED WORK

Sloppy programming may be viewed as a form of natural programming, not to be confused with natural language processing (NLP), though there is some overlap. Interest in natural programming was renewed recently by the work of Myers, Pane, and Ko (2004), who have done a range of studies exploring how both nonprogrammers 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 (1998) 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 (Bruckman & Edwards, 1999).

More recently, Liu and Lieberman have used the seminal Pane and Myers studies of nonprogrammers to reexamine the possibilities of using natural language for programming, resulting in the Metafor system (Liu & Lieberman, 2005). This system integrates natural language processing with a commonsense knowledge base, to generate “scaffolding,” which can be used as a starting point for programming. Chapter 17 discusses MOOIDE, which takes these ideas into the domain of a multi-player game. 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 (Mandelin, Xu, & Bodik, 2005). 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. The XSnippet system (Sahavechaphan & Claypool, 2006) builds on this idea to find snippets of code that return a particular type given local variables available in the current context. 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 (Ko & Myers, 2004; Ko, Myers, & Aung, 2004; Phalgune et al., 2005), studying problems faced by end users in comprehension and generation of code (Ko et al., 2004; Wiedenbeck & Engebretson, 2004) and increasing the reliability of end-user programs using ideas from software engineering (Burnett, Cook, & Rothermel, 2004; Erwig, 2005). 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.

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?

Key insight

One key insight behind sloppy programming 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 Web page. They might use a tool like Chickenfoot (see Chapter 3) which allows end users to write Web customization scripts. Now let us 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 Web page 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 time consuming 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.

Interfaces

Perhaps the simplest interface for sloppy programming is a command-line interface, where the user types keywords into a command box, and the system interprets and executes them immediately. The interface is similar to the model assumed by Web search engines: you enter a query and get back results.

Sloppy commands can also be listed sequentially in a text file, like instructions in a cooking recipe. This model resembles a typical scripting language, though a sloppy program is more constrained. For instance, it is not clear how to declare functions in this model, or handle control flow. These enhancements are an area of future research.

In other research, we have explored an interface for sloppy programming that acts as an extension to autocomplete, where the system replaces keywords in a text editor with syntactically correct code (Little & Miller, 2008). 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())” (assuming a context with the local variables “lines” and “in” of the appropriate types).

Benefits

Sloppy programming seeks to address many of the programming challenges raised in the Introduction. To illustrate, consider a query like “left margin 2 inches” in a sloppy command-line interface for 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)”, but the sloppy version of this query has several advantages.

First, note that we do not require the user to worry about strict requirements for punctuation and grammar in their expression. They should be able to say something more verbose, such as “set the left margin to 2 inches”, or express it in a different order, such as “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, as opposed to many end-user programming systems which use a graphical user interface (GUI) to provide structure for user commands. The benefits of pure text are highlighted in (Miller, 2002), 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, which tend to support only modifications explicitly exposed in the GUI. Also, text can be easily shared with other people through a variety of communication media, including Web pages, paper documents, instant messages, and email. 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 often 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, because 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 not 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. All of these systems take the form of plugins for the Firefox Web browser, and allow end users to control Web pages with sloppy commands. Two of the systems use a command-line interface, whereas the other allows users to record lists of sloppy commands.

Sloppy Web command-line

Our first prototype system (Little & Miller, 2006) 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-line interface which users can use to perform common Web browsing tasks. Important user interface elements are shown in Figure 15.1.

The functions in the Web prototype map to commands in Chickenfoot (see Chapter 3). 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 Web page to click. The “enter” command takes two arguments, one describing a textbox on the Web page, 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, such as “enter”, “pick”, and “click”, we added synonyms for these names, such as “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. Figure 15.2 is an image taken directly from the instructions provided to users during a user study. The red circles in the figure indicate changes the user is meant to make to the interface, using pictures rather than words so as not to bias the user’s choice of words. The following sloppy instructions for performing these tasks were generated by different users in the study and interpreted correctly by the system: “field lisa simpson”, “size large”, “Return only image files formatted as GIF”, “coloration grayscale”, “safesearch no filtering”.



FIGURE 15.1

The graphical user interface (GUI) for the Web command-line 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 (c).

Find results	related to all of the words	<input type="text" value="lisa simpson"/>
	related to the exact phrase	<input type="text"/>
	related to any of the words	<input type="text"/>
	not related to the words	<input type="text"/>
Size	Return images that are	<input type="text" value="large"/>
Filetypes	Return only image files formatted as	<input type="text" value="GIF files"/>
Coloration	Return only images in	<input type="text" value="grayscale"/>
Domain	Return images from the site or domain	<input type="text"/>
SafeSearch	<input checked="" type="radio"/> No filtering <input type="radio"/> Use moderate filtering <input type="radio"/> Use strict filtering	

FIGURE 15.2

Example tasks shown to users in a study. The red circles indicate changes the user is meant to make to the interface, using pictures rather than words.

However, we did discover some limitations. One limitation was that the users were not sure at what level to issue commands. To fill out the form above, they didn't know whether they should issue a high-level command to fill out the entire form, or 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-level 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 system's 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 such as this.

Another limitation was that users were not always able to think of a good way of describing certain elements within an HTML page, such as a textbox with no label. Consider, for example, the second task (the second red oval) in [Figure 15.3](#).

Street address	<input type="text" value="777 Home Drive"/>
<input type="text" value="PO Box 777"/>	
City	<input type="text"/>
State / Province	<input type="text" value="Arizona"/>
Zip / Postal code	<input type="text"/>

FIGURE 15.3

This is a more difficult example task from the study – in particular, it is unclear how users will identify the textbox associated with the second red oval using words.

We also found that, despite our best efforts to support the wide range of ways that people could express actions, we still found people trying commands we did not 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.

CoScripter

As discussed in Chapter 5, CoScripter was built to explore the synergy of combining several programming paradigms: programming by demonstration, the use of a wiki as a form of end-user source control, and sloppy programming. The newest version of CoScripter uses a grammar-based parser, but this section will focus on an early prototype of the system. CoScripter is an extension to Firefox. It appears as a sidebar to the left of the current Web page as shown in [Figure 15.4](#) (note that this is an early screenshot of the system).

The original CoScripter system experimented with additional user interface techniques to assist users with the sloppy programming paradigm. Because the sloppy interpreter was sometimes wrong, it was 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 [Figure 15.4](#). Suppose the user has selected the highlighted line: “scroll down to the ‘Shop by commodity’ section, click ‘View commodities’”. CoScripter interprets this line as an instruction to click the link labeled “View commodities”. At this point, the system needs to make two things clear to the user: what the interpreter is 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 it chose this particular action by highlighting words in the keyword query that contributed to its choice. 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 CoScripter also includes a feature to allow users to record commands by interacting with Web pages rather than writing any sloppy code. Unfortunately, the system would sometimes record an action as a sloppy command that the sloppy interpreter would then fail to

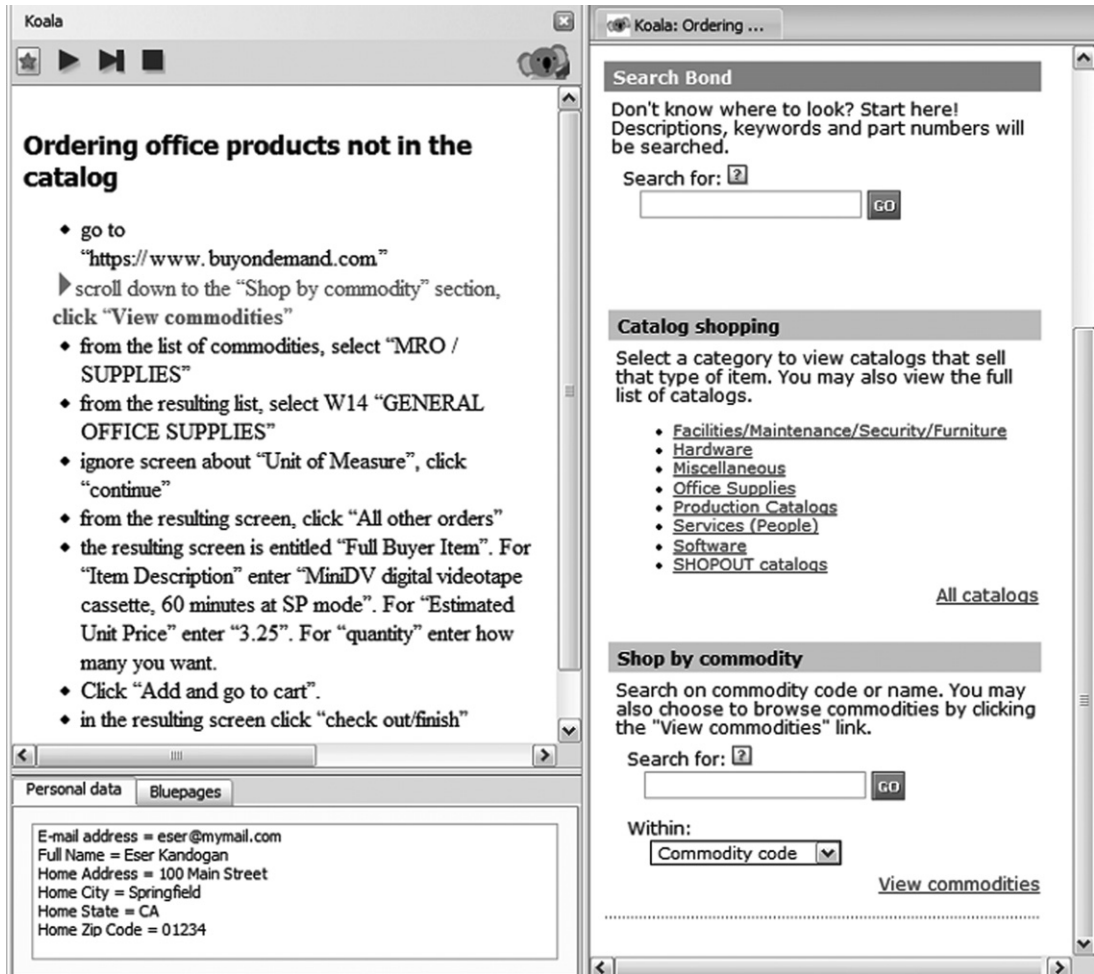


FIGURE 15.4

User interface for CoScripter: a sidebar in the Firefox Web browser.

interpret correctly. This is due to the ambiguous nature of sloppy programming, and suggests directions for future work. For instance, the recorder could notice that the sloppy command is ambiguous, and add additional keywords to alleviate the ambiguity. The newest version of CoScripter uses a grammar-based approach to parse the language, because it is so important for recorded commands to play back as expected (and for other reasons discussed in Chapter 5).

Inky

Inky (Miller et al., 2008) 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 (see Figure 15.5) 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 previously, 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 such as “32-D463” or a nickname such as “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.

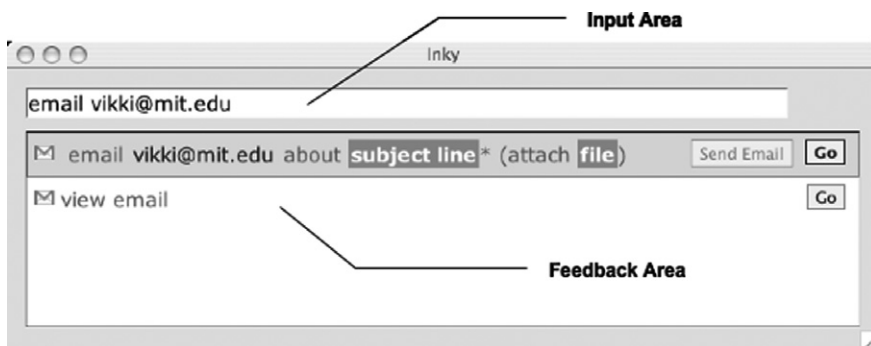


FIGURE 15.5

User interface for the Inky prototype.

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 (such as a room reservation Web site) 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 (see [Figure 15.6](#)). 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.

The visual cues of the interpretation were designed to make it easier to scan. A small icon indicates the Web site that the function automates, using the favicon (favorites icon) 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 [Figure 15.7](#), “name, email, office, etc.” is a missing argument with no default, whereas “this month” is an argument which defaults to the current month.


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

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

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

FIGURE 15.6

Example inputs and their corresponding interpretations using the Inky prototype.



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

FIGURE 15.7

Example outputs showing the different colorations used to indicate missing arguments.

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 [Figure 15.7](#), 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 with parentheses, such as the “(repeats never)” argument in the figure. 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.

ALGORITHMS

All of the systems described above need to convert a sloppy command 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 are limited mainly by what is actually possible in the current context. This is the case in a typical form-based Web page, where the number of commands is roughly proportional to the number of form widgets. This algorithm was used in the early CoScripter system.

The second approach assumes that the space of 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 algorithm that was 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. These improvements were used in the Inky prototype.

CoScripter algorithm

We describe this algorithm in three basic steps, using the example instruction “type Danny into first name field” on the simple Web form shown in [Figure 15.8](#).

First, the interpreter enumerates all the possible actions associated with various HTML objects in the document, such as links, buttons, textboxes, combobox options, checkboxes, 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 in Figure 15.8 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 extract arguments from the sloppy command (e.g., “Danny”). The key idea of this process is to extract a subsequence from the sloppy command, such that the remaining sequence is still a good match to the keywords associated with the Web object, and the extracted 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, we extract the subsequence “Danny” (which gets points for being followed by the preposition “into” in the original sloppy command). The remaining sequence is “type into first name field”, which still has all 4 keyword matches to the text field.

This technique is surprisingly effective, because a Web page provides a very small search space.



FIGURE 15.8

Example of a simple Web form.

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, such as 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 such names as “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; for example, 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, such as the copy command in most operating systems. In this case, the idea would be to name one argument “from”, and the other argument “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 lowercase letter followed by an uppercase 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” and “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 lowercase 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 lowercase, and then passed through a common stemming algorithm (Porter, 1980).

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 command restricts the return type).

For example, consider that the user types the sloppy query:

```
search textbox enter Hello World
```

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. One function in the Web command prototype is called “enter”, and so the algorithm considers dividing the sequence into three substrings as follows:

```
search textbox      (enter function: enter)      Hello World
```

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.

In our example, the “enter” function has two arguments: a string and a textbox to enter the string into. One natural possibility considered by the algorithm is that the first substring is one of the arguments, and the third substring is the other argument:

```
(argument: search textbox)      (enter function: enter)      (argument: Hello World)
```

Then, for each set of n substrings, it considers every possible matching of the substrings to the n arguments. For instance, it might consider that the left argument is the second argument to the enter function, and that the right argument is the first argument:

```
(argument 2: search textbox)      (enter function: enter)      (argument 1: Hello World)
```

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.

For instance, a recursive call to the algorithm will consider the substring:

```
search textbox
```

The desired return type for this argument will be Textbox, and one function that returns a textbox is called “findTextbox”. This function takes one argument, namely a string identifying the label of the textbox on the Web page. The algorithm will then consider the following breakdown of this substring:

```
(argument 1: search)      (findTextbox function: textbox)
```

The algorithm will ultimately resolve this sequence into:

```
findTextbox("search")
```

These resulting function trees are grafted as branches to a new tree at the previous level of recursion, for example:

```
enter("Hello World", findTextbox("search"))
```

The system then evaluates how well this new tree explains the token sequence – for example, how many nodes in the tree match keywords from 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

Intuitively, we want to uncover the function tree that the user intended to create. All we are given is a sequence of tokens. Therefore, we want to take each function tree and ask: “if the user had intended to create this function tree, would that explain why they entered these tokens?” Hence, we arrive at the notion of *explanatory power* – how well a function tree explains a token sequence.

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 (because 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 could be learned from a corpus of user-generated sloppy inputs.

Algorithm optimization

The algorithm described above slows down exponentially as more types and functions are added. The current Inky prototype only has 11 types and 35 functions, but the system is meant to support many user-supplied types and functions. In order to deal with the larger scale, we have explored a way of making the algorithm more efficient, at the expense of making it less accurate.

The algorithm in Inky uses a variant of the algorithm described in (Little & Miller, 2008). Briefly, this algorithm uses dynamic programming to find the fragment of the user's input that best matches each type known to the system. The initial iteration of the dynamic program runs the type-matching recognizers to identify a set of argument types possible in the function. Subsequent iterations use the results of the previous iteration to bind function arguments to fragments that match each argument's types. Interpretations are scored by how many input tokens they successfully explain, and the highest-scoring interpretations for each type are kept for the next iteration. For example, three integers from a previous iteration may be combined into a date. The dynamic program is run for a fixed number of iterations (three in the Inky prototype), which builds function trees up to a fixed depth. After the last iteration, the highest-scoring interpretations for the void return type (i.e., functions that actually do something) are returned as the result.

DISCUSSION

The prototypes and algorithms discussed previously work well in many cases. All of these systems have been tested with user studies, and have demonstrated some level of success. CoScripter has grown beyond a prototype, and has many thousands of real users, though the sloppy interpreter had to be refined to be less sloppy. Hence, there is still work to be done in the field of sloppy programming. In this section, we discuss a number of limitations, and directions for future work.

Limitations

There are three primary limitations of sloppy programming: consistency, efficiency, and discoverability. Sloppy commands are not always interpreted the same way. The algorithm and heuristics in the sloppy interpreter may change, if the user gets a new version of the software. This happens with conventional scripting languages too, but the consequences may be more difficult to notice with sloppy programming, because the system will try to interpret commands as best it can, rather than halt with a syntax error if a command is broken. (CoScripter has some heuristics to notice if a command is broken.) Also, the context of the command can change. If the user is executing a

command in the context of a Web page, then the interpretation of that command may change if the Web page changes. Finally, a user may modify a sloppy command without realizing it; for example, they may invert the order of keywords. This will usually not have an effect, but depending on the algorithm and heuristics used, it can have an effect.

The next limitation is efficiency: there is a tradeoff between interpreting a command accurately, and interpreting a command quickly. The Web command-line algorithm described previously appears to interpret sloppy commands accurately, but it is extremely inefficient. It is fast enough in the Web command prototype, because the command set is very small, but the algorithm quickly becomes slow as more commands are added. We discussed an optimization to this algorithm in the previous section, but this optimization sacrifices the accuracy of interpretations by ignoring information (namely word order). Tradeoffs such as this are not yet well understood.

The final problem, discoverability, is a problem that plagues all command-line interfaces. It is not clear what the user may type. We try to overcome this by implementing our prototypes in a domain that users are familiar with, and try to have good coverage of that domain, so that the system is likely to recognize anything that the user types. However, users are very creative, and always come up with commands that designers of a system did not anticipate. Also, it is important to find ways to make the system more discoverable in domains where the user is not as familiar with the set of possible commands.

Future work

The primary goal of future work is to overcome the limitations mentioned earlier. To make sloppy programming more consistent, there are two main approaches. One is to have a fallback mechanism where users can, in the worst case, write commands in a verbose but unambiguous manner. The other approach is to fine-tune the existing heuristics, and add new heuristics. Fine-tuning heuristics will likely require a corpus of user-generated sloppy commands, along with the correct interpretation for each command. Discovering new heuristics may be more difficult, requiring creativity and testing. Validating new heuristics will again benefit from a corpus of sloppy commands.

The first step toward making sloppy programming more efficient may simply involve implementing the algorithms more efficiently, possibly using lower-level languages to achieve greater speed. The second step involves gaining a better understanding of the heuristics used in the algorithms. Some optimizations may require ignoring certain heuristics, and concentrating on others. These decisions would benefit from understanding the contributions that different heuristics make to the resulting interpretation quality.

We believe discoverability in sloppy programming can be enhanced in a number of ways. For instance, the Inky prototype explores the idea of showing people multiple interpretations as the user is typing. This is similar to Google showing query suggestions as the user types, based on popular search queries beginning with the characters entered so far. An extension to Inky, called Pinky (Miller et al., 2008), explores an interesting extension to this idea, where the user can interact with the suggestions themselves. These changes are propagated back to the sloppy command, allowing the user to learn more about the command language. This idea suggests a hybrid between command-line and GUI systems, with benefits from each. A side benefit of showing more information about how commands are interpreted is that users may understand the system better, and be able to write commands that are more robust to changes in the system. These ideas require future work to explore and verify.

SUMMARY

The techniques 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.

Acknowledgments

We thank all the members of the UID group at MIT and the User Interface group at 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.

This chapter is based on several earlier works:

1. A paper that originally appeared as “Translating Keyword Commands into Executable Code,” in *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (UIST 2006), © ACM, 2006. <http://doi.acm.org/10.1145/1166253.1166275>,
2. A paper that originally appeared as “Koala: Capture, Share, Automate, Personalize Business Processes on the Web,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI 2007), © ACM, 2007. <http://doi.acm.org/10.1145/1240624.1240767>, and
3. A paper that originally appeared as “Inky: a Sloppy Command Line for the Web with Rich Visual Feedback,” in *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology* (UIST 2008), © ACM, 2008. <http://doi.acm.org/10.1145/1449715.1449737>.

INKY

Intended users:	All users
Domain:	All Web sites, but a programmer must add support for each site
Description:	Inky allows users to type Web-related commands into a textbox, and it attempts to execute them. The system is not inherently restricted to any particular set of commands, but it only understands commands for which implementations have been written. Some examples include checking the weather and sending an email.
Example:	A user could use the system to add an event to their calendar with a command like "meeting with David at 4pm on Tuesday". This command would fill out the appropriate form on Google Calendar, and the user could ensure that the information was correct before clicking Submit.
Automation:	Yes, the system automates tasks typically consisting of one to three Web forms.
Mashups:	Possibly. Supported tasks may include entering information into multiple Web sites.
Scripting:	Yes. The system allows users to write commands in a sloppy scripting language, with the intent of trying to interpret anything the user types as a valid command.
Natural language:	No. The system tends to be more lenient than a classical natural language parser, because it does not require the user to use proper grammar. Writing in a very verbose natural language style is likely to confuse the system.
Recordability:	No.
Inferencing:	Yes, Inky uses many heuristics similar to natural language processing to translate queries into executable code.
Sharing:	No.
Comparison to other systems:	This system is similar to other systems that use sloppy programming, including the sloppy Web command-line prototype and CoScripter.
Platform:	Implemented as an extension to the Mozilla Firefox browser.
Availability:	Not currently available.