Programming with Keywords

by

Greg Little

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2007

Certified by . . . .
Robert C. Miller
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Programming with Keywords

by

Greg Little

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science

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, they may be able to write a set of keywords expressing a command in that domain. For instance, in the web browsing domain, a user might enter the keywords **click search button**. This thesis presents several algorithms for translating keyword queries such as this directly into code. A prototype of this system in the web browsing domain translates **click search button** into the code `click(findButton("search"))`. This code may then be executed in the context of a web browser to carry out the effect. Another prototype in the Java domain translates **append message to log** into `log.append(message)`, given an appropriate context of local variables and imported classes. The algorithms and prototypes are evaluated with several studies, suggesting that users can write keyword queries with little or no instructions, and that the resulting translations are often accurate. This is especially true in small domains like the web, whereas in a large domain like Java, the accuracy is comparable to the accuracy of writing syntactically correct Java code without assistance.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor

Acknowledgments

A special thanks to all the administrative angels who have cut through red tape on my behalf, including Marilyn Pierce, Sally Lee, Lisa Bella, Maria Rebelo, and Lisa Cole.

I'm not consciously aware of most of what my brain does, but I'm pretty sure it's just randomly trying out combinations of ideas it has stolen from its environment. The thought processes I am aware of tend to be pretty obvious and mechanical manipulations of ideas coughed up from my subconscious.

Anything that resembles creativity or insight in this thesis is more likely the result of one or more of the following influences: my adviser Rob Miller, and all the members of the User Interface Design group; Max Goldman; my mentors at IBM Almaden, Allen Cypher and Tessa Lau, and all the members of the Koala group; my mentors at Arizona State University, Sethuraman Panchanathan, John Black and Goran Konjevod, and all the members of the CUBiC lab; Sreekar Krishna; my teachers at MIT and ASU; David Karger and Leonard Faltz; my parents and family; my friends; my coworkers; my pets; Virus; strangers I've met, media I've consumed, places I've lived, physical objects I've interacted with; the stuffed killer whale on my desk; and many things I'm unaware of or have forgotten to mention.

This work was supported in part by the National Science Foundation under award number IIS-0447800, and by Quanta Computer as part of the TParty 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, or of the people, organizations and objects mentioned above.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation & Problem

When a user enters “kiler whale” into a web search engine, they do not expect it to return an error like: `Unexpected token ‘kiler’`. In fact, they expect the search engine to return hundreds of results where people have misspelled “kiler” in the same way they did. In addition, they expect the search engine to suggest the proper spelling of “killer whale”.

In effect, users expect the search engine to try everything within its power to make the user’s input the correct input. The burden is on *it* 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 forgot to put quotes around “hello world”. 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 makes 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 typically use formal languages, requiring users to cope with rigid and seemingly arbitrary syntax rules for forming expressions. The canonical example is the semicolon required at the end of each expression in C, but even modern scripting languages like Python and JavaScript have similar arbitrary syntax requirements.

Not only do scripting interfaces use formal languages, but different applications often use *different* formal languages. These languages are often similar—usually 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.

To help alleviate these problems, I propose that instead of returning a syntax error, compilers should act more like a web search engine. They should first search for a syntactically valid expression over the scripting language and API. Then they should present this result to the user for inspection, or simply execute it, if the user is feeling lucky.

1.2 Keyword Programming

This is the essence of keyword programming. The user should be able to enter keywords, and the computer should try everything within its power to make this input the correct 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, "now click the search button" shares certain features with the Chickenfoot [5] code `click(findButton('search'))`. In particular, a number of keywords are shared between the natural language expression, and the programming language expression.

This thesis hypothesizes 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. One reason this may be true is that language designers, and the designers of APIs, often name functions such that they read like English. This makes the functions easier to find when searching through the documentation.

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, this thesis shows that it is possible to approximate this search to varying degrees, and that the results are often useful.

1.3 Interface

This thesis proposes two primary interfaces for keyword programming. The first is a command line interface, where the keywords supplied by the user are executed when they press enter. Note that although commands traditionally require a verb, the keyword queries may be suggestive, without using an imperative verb. For example, consider the keyword 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 keyword command line system would 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 keyword completion interface would let them enter:

add line

and the system would suggest something like:

```
lines.add(in.readLine());
```

1.4 Benefits

Keyword programming addresses many of the issues raised previously. To illustrate, consider the example of **left margin 2 inches**. First, note that the user did not need to worry about strict requirements for punctuation and grammar in their expression. For instance, they could have said something more verbose, like **set the left margin to 2 inches**, or they could have expressed themselves in a different order with **2 inches, margin left**. Second, the user did not need to know the syntactic conventions for method invocation and assignment in Visual Basic. The same keywords would work regardless of the underlying scripting language. Finally, the user did not have to search through the API to find the exact name for the property they wanted to access. They also did not need to know that `LeftMargin` was a property of the `PageSetup` object, or that this needed to be accessed via the `ActiveDocument`.

Another advantage of keyword programming is that it accommodates *pure text*. The benefits of pure text are highlighted in [23], 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 keyword queries (a keyword 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. Keyword programs offer the same ability for arbitrary applications—you can store or share a keyword 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**).

Keyword programming also offers benefits to expert programmers as an extension to autocomplete, by decreasing the cognitive load of coding in several ways. First, keyword 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.

1.5 Specific Contributions

This thesis presents four algorithms for translating keyword queries into executable code: Reference, Koala, Keyword Tree, and Bottom-up. Each has tradeoffs between speed, accuracy, and domain specificity. The Reference algorithm is a brute force algorithm, and may be viewed as a reference algorithm that the other algorithms attempt to approximate. The Koala algorithm is designed to work in domains where the number of possible actions available to the user at any given point is relatively small. The Keyword Tree and Bottom-up algorithms are designed to support keyword queries in a general purpose programming language (specially Java).

Four prototypes are also presented. The first two prototypes use the Reference algorithm. One is a command line interface for the web, which translates keyword commands into Chickenfoot [5] commands. These commands allow users to navigate to web pages and interact with web forms. The other prototype is a command line interface for Microsoft Word, which translates keyword commands into Visual Basic commands. These commands can be used to access document properties and issue commands like *save* and *print*.

The third prototype is a system called Koala, which I helped develop at IBM. Koala enables end users to record and automate their interactions within the Mozilla Firefox web browser. As users interact with the browser performing a process, Koala records all the forms filled, links and buttons clicked, and menus selected in a script. Instructions in the script are saved as keyword commands, allowing them to be easily read, understood, and modified by users. Moreover, the algorithm that Koala uses to interpret keyword commands takes advantage of the fact that web pages offer a limited number of possible user actions.

The fourth prototype is an Eclipse Plug-in that allows users to perform keyword query completions in their Java source code. It can use either the Keyword Tree or Bottom-up algorithm, each of which is designed to work efficiently in environments with thousands of functions.

To evaluate the prototypes, I used both user studies and automated tests on an artificial corpus. One user study operates in the web domain, and has users enter keyword commands to complete tasks on various web pages. Another user study operates in the Java domain (aimed at programmers familiar with Java), and asks users to enter keyword queries to complete a snippet of Java code. The automated tests also operate in the Java domain on keyword queries generated artificially by taking actual expressions in open source projects and obfuscating them.

1.6 Outline

In the next chapter, I position keyword programming in the larger field of end-user programming research. In chaper 3, the user interface for keyword programming is presented. This is followed with a formal model for describing the problem in chapter 4, after which I discuss the algorithms in chapter 5. The prototypes are then presented in chapter 6, followed by evaluations of the prototypes in chapter 7. Finally, I discuss the results of the evaluations in chapter 8, and draw conclusions and present directions for future work in chapter 9.

Chapter 2

Related Work

The goal of keyword programming is to make code generation easier, with the hope of making programming less cumbersome and more accessible. This chapter attempts to position keyword programming in the broad field of research with similar goals.

The difficulties of programming are most apparent in novice programmers learning how to program, and in end-users trying to use scripting interfaces for applications; hence, most of the research in this area focuses on these two use cases, and the field of research is often termed *end-user programming*, though my research also aims to help expert programmers.

The goal of end-user programming research is extremely difficult. Ultimately, the computer should simply understand anything the user says, which is tantamount to passing the Turing Test [31]. In such cases, there are often two broad strategies for tackling the problem: top-down, and bottom-up.

In the top-down approach to end-user programming, researchers study humans trying to communicate to computers, and try to infer principles to guide the development of programming systems. In the bottom-up approach, researchers build a prototype system, and see how well it conforms to user's needs.

2.1 Top-down Principles

Top-down researchers thus far have established a number of principles for end-user programming systems that developers may use to inform the design of programming interfaces. My goal in exploring this research is to argue that keyword programming adheres to these principles, in as far as they are applicable.

Ko, et al., attempt to categorize the issues that end-users commonly face when using programming systems in [17]. They focus on the programming system as a whole, as opposed to focusing on a particular language, though their tests are from students using VB.NET 2003. The "barriers" they saw students face include difficulties in: knowing where to start (design), finding the right tool (selection), making tools work together (coordination), figuring out how to use a tool (use), figuring out why they are getting an error (understanding), and knowing how to test hypotheses (information).

Keyword programming strives to ease the barriers in design, selection, coordination and use. It focuses particularly on cases where the user is familiar with the capabilities of the system, but does not know how these capabilities map into the API or the syntax of the language. For instance, a user may know that buttons on a web page can be clicked, but they do not know which of the following commands does the job: `click(find(BUTTON, "submit"))`, `(click (get-button 'submit))`, `getButton("submit").click()`. Keyword programming allows the user to enter **click submit button**, and have the system explore the API for a reasonable match, and render this match in the appropriate language syntax.

Green proposes a vocabulary for describing common programming language design issues, which he calls cognitive dimensions [14]. The idea is that they would act as a sort of checklist of issues to consider when validating a new design. They also provide a vocabulary of discourse for designers to communicate amongst themselves. One of these dimensions is "Hidden dependencies", which refer to relationships between objects in a language (or spreadsheet or word processor style sheet, etc...) that are not made explicit to the user. For instance, the cells that depend on a certain cell in a spreadsheet are not marked in many spreadsheet programs. Another dimension is “Viscosity”, which refers to how difficult it is to make certain changes. For instance, to insert a new object in a hierarchy in Java requires changing many lines of code so that objects extend the new inserted object. The idea is that new language designers should keep in mind the various dimensions, and make informed choices regarding the tradeoffs given the goals of their language. Another motivation behind cognitive dimensions stems from the recognition that user studies are expensive and time consuming to run, and these dimensions can be used to alleviate some of the necessity for user studies.

These issues are important to keep in mind when considering using keyword programming as a language, where a program would consist of a list of keyword queries that are translated into formal scripting commands at run-time. One fear is that long keyword programs would contain many hidden dependencies, and tend to be very viscous. Hence, this thesis explores the temporal interfaces of a command line and autocomplete. The Koala prototype does use keyword queries as the final program representation, but the programs tend to be very short, and they are completely linear—no control flow is supported.

Pane, et al., conducted studies in [24] to explore common ways that end-users program when they are not given any constraints, i.e., they are allowed to write free-form text and/or draw pictures. Some notable observations include: event based programming is common; the keyword AND is often used for sequencing in natural language, and when it is used for Boolean logic, it is often used more like the logical-OR; the keyword THEN is used for sequencing much more often than in IF/THEN style statements; loops are usually expressed implicitly with set-theoretic constructs like plurals; in general, natural language programs are imprecise (or wrong) in cases like expressing a range. The study advocates using constructs in programming languages that map more closely to natural language.

Unfortunately keyword programming does not take advantage of most of these observations in itself, since it relies heavily on the structure of the underlying scripting language and API. However, it would be interesting to study what improvements could be made to scripting languages and APIs based on these principles, given that keyword programming will be used as a layer between the user and the API.

2.2 Bottom-up Prototypes

Button-up researchers have built many prototypes of programming interfaces, and tested their effectiveness on novice programmers, end-users and even expert programmers. My goal in this section is to argue that keyword programming is a novel interface, and is likely to yield benefits for users not found in existing systems.

Kelleher, et. al, provide various high-level strategies employed by many prototypes in [16]. A few of these include: Make the program always active and inspectable; Allow the user to demonstrate what they want the computer to do, rather than writing it out; Use structured editors which enforce the creation of syntactically correct programs; Make the language more like English; Build collaboration mechanisms into the programming environment.

Spreadsheets are the canonical example of making a program always active and inspectable, and spreadsheets comprise a significant portion of end-user research. However, spreadsheets still rely on commands written in cells, and these commands still have the same rigid syntax requirements of any programming language. Keyword programming focuses on allowing users to generate commands in any language, and so from this point of view, spreadsheets are simply another language. The key problem then is generating commands, and many prototypes focus on making it easy for users to specify what they want the computer to do.

2.2.1 Programming by Demonstration

The notion of allowing the user to demonstrate what they want the computer to do, rather than writing it out, is embodied by the notion of programming-by-demonstration or PBD (e.g. [10, 1]). The basic idea behind PBD is to infer a program from manual actions taken by the user, like a macro recorder.

A modern example of this technique is DocWizards [3]. This system allows user to create *follow-me documentation wizards* by demonstration. As the name implies, follow-me documentation wizards are a form of documentation that behaves like a wizard in the sense that the system highlights the next action in the interface, and has the ability to perform this action for the user. However, unlike a traditional wizard, follow-me documentation wizards allow the user to deviate from the script.

Active tutorials of this form are intuitively more useful than traditional documentation, since the system can highlight the next action in the actual interface. However, even traditional tutorials or documentation are difficult to author, especially when screen shots are required. DocWizards overcomes this hurdle by allowing the user to perform the actions as they would normally, and record the actions using hooks in the application.

One challenge users face with this and many other programming-by-demonstration systems is editing scripts. Although the interface allows users to delete and move script actions, they cannot change script actions. Allowing users to change actions would be difficult, since the system would need to expose the underlying scripting language, which is bound to be less readable than the current implementation.

This is where keyword programming may prove useful, as a method of displaying recorded actions that *can* be edited, and even authored without demonstration. The new actions would pass through the keyword programming translation algorithms into the underlying scripting language.

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

2.2.2 Structured Editors

Structured editors are an alternative approach toward the problem of giving users access to a formal scripting language. Structured editors like [8, 15, 16, 22, 30] can be a good approach for end users to write and edit scripts, since they force the script to retain a syntactically correct form. They also often provide menu systems so the user can recognize, rather than recall, commands. For example, Apple Automator [8] provides a menu listing the available application commands, and each command's parameters can be configured by a graphical user interface.

Kelleher mentions two benefits of structured editors while describing the Alice2 system [15]. The first benefit is that users are prevented from making syntax errors, which can be frustrating for novice programmers. The second benefit is that because users build expressions from tiles in a menu, this same menu allows users to explore all the capabilities of the system.

One motivation behind using a structured editor is that users complained about having to remember syntax in the prior version of Alice [9]. This is a problem that keyword programming also addresses. Of course, users also complained about typing in general. Keyword programming still requires typing, but not as much (since keyword queries tend to be shorter than syntactically correct programming expressions). It is also unclear whether users would complain as much about typing if syntax errors were impossible, as they are in keyword programming.

The system described in [11] is an interesting application of structured-editors to spreadsheets. The idea is to make developing a spreadsheet into a two phase process. The first phase involves developing a template for the spreadsheet (using the Visual Template Specification Language, ViTSL). The template specifies how and where data may be entered. The template also contains all the formulas for the spreadsheet.

The second phase consists of adding data to the spreadsheet (using an Excel extension called Gencel). This may involve adding columns and rows, and adjusting the value of cells, but all of these operations are constrained to follow the template. The template specifies where columns and rows may be inserted (and provides legal default values for all the inserted cells). The template also specifies the type of data that each cell contains, and the user may not enter data of an invalid type. User studies suggest that users value the constraints enforced by the system when they are using a template, but it is left to future work to develop a system whereby users may create their own templates.

These systems have had some success, but one problem with structured editors is that they tend to require the user to think in a certain order, which can be counter-intuitive. For instance, consider the example in the previous section where the user enters the keywords **3 columns** into a word processor. If the user wants to express the command **3 columns** in a structured editor, they need to begin by filling in the first slot in an expression builder, and this slot is unlikely to accept the number 3. In fact, they may not know whether to begin with an assignment template or a function template, depending on how this particular property is set in the API. The main point here is that structured editors require some planning on the part of the user toward building their expression, even if they can pick out the pieces with menu systems. Keyword programming relaxes this restriction.

2.2.3 Natural Language Programming

One approach that seeks to overcome some of these problems is natural language programming, i.e., making English a programming language. Sammet advocated the use of English (or any natural language) as a programming language as early as 1966 [29]. However, interpreting arbitrary English expressions as executable code has proven to be a real challenge. A study by Miller [21] outlines some of these challenges, including the variety of styles humans use to express ideas.

Because of this, Liu argues in [19] that natural language interpretation falls into two categories: building a formal grammar, and using information retrieval techniques. Formal grammar based systems like NLC [2] use formal languages that read like English. The problem with these systems is that they do not permit grammatical errors or extraneous words. Liu's information retrieval techniques include small template grammars that match pieces of a natural language expression, and they also include treating the entire input as a list of keywords in a database query. Template based systems like NaturalJava [27] try search for recognizable language constructs within an expression. However, these systems still require the user to incorporate such constructs into their expressions. Keyword programming falls into the category of treating the input as a sequence of keywords used as a database query.

Liu also presents a general mapping of some natural language constructs to programming constructs. These mapping include the idea that a noun is an object, an adjective is a member variable, and several linguistic features represent if/then statements and loops. They also recognize that humans often use set-theoretic style statements instead of loops (like “the good people”, instead of “for every person, include that person if they are good”).

These concepts are explored in Metafor [18], which allows a human to write a story-style description of a program, and Metafor automatically generates “scaffolding” for the code. This system uses a natural language parser called MontyLingua, which translates sentences into VSOO (verb-subject-object-object) form. MontyLingua takes advantage of a database of common sense knowledge to aid in parsing.

An example of a system more similar to keyword programming is the Natural Query Markup Language (NQML) system [25], which is designed to allow users to enter natural language queries which are translated into SQL. NQML itself is an XML specification of the metadata used to map a database into natural language, consisting of grammar templates and synonyms. The system consists of algorithms for performing the translation of a natural language query into SQL given this data. Keyword programming also uses synonyms, and it can be thought of as using grammar templates, though these are extremely loose, and are derived automatically from the API itself.

A couple commercial products similar to keyword programming include Quicksilver [4] for the Mac, and the “quick add” feature in Google Calendar [13]. Quicksilver is a program launcher, but the interface is textual— as the user enters text, the program suggests commands that they may intend to execute based on the keywords they have entered so far. The “quick add” feature of Google Calendar is a command line interface for adding events to a calendar. It uses natural language processing to parse inputs, but seems to be quite robust to different styles of expressing event details. The details of the algorithms behind these systems are not exposed, so it is not clear how similar they are to keyword programming at an algorithmic level, but they seem to have the same goal in mind of trying to make sense of anything the user types.

2.2.4 Collaboration

An alternative to the previous line of reasoning is, rather than focus on making the language easy to learn, focus on making it easy for end-users to share their knowledge and help each other.

Bruckman's MooseCrossing [6] is a programming system aimed at children that uses formal syntax resembling natural language. In this case, the goal of the research is to study the ways that children help each other learn to program in a cooperative environment.

The Koala system presented in this thesis attempts to draw from the power of collaboration by storing scripts on a wiki, so that different people can run and even improve each other's scripts. Of course, Bruckman found that almost half of errors made by users were syntax errors, despite the similarity of the formal language to English [7]. Hence, Koala stores scripts as a series of keyword commands, each of which has no syntax.

2.2.5 Helping Expert Programmers

Most of the systems described so far focus on end-users and novice programmers, whereas this thesis also aims to benefit expert programmers. When it comes to expert programmers, various techniques have been tried to reduce the effort spent forming expressions. Autocomplete is a good example. A couple more recent examples include Prospector [20] and XSnippet [28]. Both of these systems work in the Eclipse IDE on Java code.

Prospector is a system that helps users generate code that will return a desired type. Note that the core system asks for an input type, but in practice, the system will run queries on all available input types given the current scope. The system generates code based on method signatures in the API, as well as example code. Example code is used to figure out which downcasts are legal (since legal downcasts are difficult to infer based on method signatures). Heuristics are used to rank the resulting solutions, the primary heuristic being short length.

XSnippet is a system for returning relevant snippets of code based on a desired object type, and the context of available types (in the form of parameters and member variables). The system also explores several heuristics for ranking the returned snippets, including length, frequency of use, and some measure of how well the context of the snippet matches the context of the query.

Keyword programming as a form of autocomplete is similar to both of these systems, but differs in a couple respects. First, the keywords in a query provide more information than the available types, so a keyword query translation algorithm has more information with which to prune the search space. This allows users to build expressions that return `void`, and do not accept any types from the current context as input (e.g. the leaves of the expression tree may be functions, as opposed to variables). The second difference is that the translation algorithms presented in this thesis do not rely on code snippets; rather, the algorithms generate code based on the method signatures in the API. This can be an advantage over XSnippet, when there is no good corpus of snippets for an API, but the best approach is probably the one taken by Prospector which uses both generated code and snippets.

Chapter 3

User Interface

At its core, keyword programming is a user interface designed to make it easy for humans to suggest programmatic instructions to the computer. Toward this end, the user must have some idea of the capabilities of the system, along with the conventional vocabulary used to describe these capabilities (e.g. the white-space surrounding a document is called the "margin"). Beyond this, our hope is that the system is natural and intuitive. That is, a user should not have to read this chapter in order to *do* keyword programming.

3.1 Functions

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

The user can also invoke a function without naming it, merely by including its arguments. This is a sufficient suggestion in those cases where there is only one function which is likely to take the given arguments. For instance, consider **MIT**

into the search textbox. This suggests entering "MIT" into a textbox labeled "search", even though we didn't explicitly say **enter**. The web domain prototype understands this expression because it has only one command that accepts a string and a textbox (namely `enter`).

3.2 Basic Data Types

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

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

The exact set of basic data types depends on the domain of the interpreter. The web domain prototype includes types for integers, strings, booleans, URLs, and keyword-lists (which identify objects in the webpage). The Word domain prototype includes types for ints, longs, booleans, and strings. The Eclipse Plugin prototype does not currently include basic data types, though these are not difficult to add.

3.3 Identifying Arguments

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

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

3.4 Nested Functions

The system supports nested functions. For instance, **pick the 4GB RAM option** may translate into `pick(findOption("4GB RAM"))`. Note the nested invocation of `findOption`. Two of the algorithms—namely the Reference and Keyword Tree algorithms—place a phrase structure restriction on nested functions; in particular, a nested function can only be formed from a contiguous portion of the keyword query. This example obeys this restriction because **4GB RAM option** is a contiguous string within the query. We could have also exchanged the order to get **option 4GB RAM**, but we could not split up the subexpression to get **option pick 4GB RAM**.

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

Even without this fallback mechanism, this restriction does not appear to be prohibitive. No users in our study of the web based prototype formed expressions violating this rule (and users correctly formed 31 expressions which could have violated the rule). Also note that some of the algorithms presented do not enforce this restriction at all— not enforcing the restriction generally makes the algorithms faster, but at a cost of limiting the user's expressive power.

3.5 Resolving Ambiguity

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

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

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

The second technique for resolving substring ambiguity is to include argument names to identify an argument, as discussed earlier. Using this technique, the user could say **enter binary into search textbox**, and the system would favor an interpretation where the subexpression **into search textbox** was treated as the `into` argument of the `enter` command.

When these techniques fail (or the user fails to employ them), the system is left with an ambiguous expression. In such a case, the system can present the most likely candidates to the user for inspection. This is discussed more in the *Feedback* section below.

3.6 Extraneous Words

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

3.7 Feedback

In the case of a command line interface, where the keyword queries are executed immediately after the user presses enter, it is useful to let the user know what action the computer has taken. One method is to supply a graphical indication. For instance, when a user clicks a link in a webpage using a keyword command, our web prototype animates a green translucent rectangle over the link. This assures the user that the correct link was chosen.

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

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

In the case of an autocomplete interface in an IDE, the feedback comes in the form of generated source code, which an expert programmer can inspect to make sure it what they want. Again, in the case of ambiguous queries, it may be useful to display several possible expressions for the user to choose from, before replacing the keyword query with the generated code.

Chapter 4

Model

The user interface described in the previous chapter depends heavily on context. The user's keyword query in one domain may have a different interpretation in another domain, depending on the functions exposed through the API, and the state of the application.

The algorithms presented in this thesis attempt to map keyword queries to syntactically valid expressions given this context. In order to make this problem more well defined, it is necessary to provide a model for the context.

The model serves as a layer of abstraction over the underlying programming language and environment. This allows the algorithms to be presented in a more generic manner.

The model M presented here is the triple (T, L, F), where T is a set of types, L is a set of labels, and F is a set of functions. The purpose of defining types, labels and functions is to model programming language expressions that the user may intend with their keyword query.

4.1 Type Set: T

The first part of the model is a set of types T. Each type is represented by a unique name. In the web prototype, types include html objects like `Textbox`. In the Eclipse Plugin, types come from Java's fully qualified names for the types. Examples include `int` and `java.lang.Object`.

We also define $sub(t)$ to be the set of both direct and indirect subtypes of t. This set includes t itself, and anything assignment-compatible with t. We also include a universal supertype $\top$, such that $sub(\top) = T$. This is used when we want to place no restriction on the resulting type of an expression, e.g., when the expression is not nested inside any other expression in Java. Note that `java.lang.Object` is *not* the same as $\top$; in particular, basic data types like `int` are not subtypes of `java.lang.Object` in Java.

4.2 Label Set: L

The second part of the model is a set of labels L. Each label is a sequence of keywords. Labels are used to represent function names, so that they can be matched against the keywords in a query.

Note that the algorithms presented in this thesis treat labels as *bags* of keywords, but they are modeled as sequences so as not to throw away word order information. In principle, word order information may serve as an additional heuristic for future work.

Functions can have many keywords, usually consisting of synonyms for the function name. For instance, the `enter` command in the web prototype—used for entering text into form fields—has labels like `type`, `write`, `insert`, `set`, and the `=` symbol.

Basic data types are a special case, since they have an infinite number of synonyms, e.g., an integer would have the labels: `1`, `2`, `3478`, `-90000`, etc... This is fine as far as mathematical models are concerned, but it presents a problem in practice. To deal with this problem, the algorithms generally handle basic data types differently than normal functions, often with regular expressions like "[0-9]+".

Arguments can also have labels, which may include prepositions naming the grammatical role of the argument. Our model doesn't support argument labels explicitly, though we can represent them as functions which take one parameter, and return a special type that fits into only the parent function.

The Eclipse Plugin gets function labels from Java functions by breaking up the names at capitalization boundaries. For instance, the method name `currentTimeMillis` is represented with the label (**current**, **time**, **millis**). Note that capitalization is ignored when labels are matched against the user's keywords.

4.3 Function Set: F

The final part of the model is a set of functions F. Functions are used to model all programmatic constructs in an expression. In the web prototype, these functions generally represent Javascript functions from Chickenfoot [5]. In the Eclipse Plugin, functions represent each component in a Java expression that we want to match against the user's keyword query, including methods, fields, and local variables.

We define a function as a tuple in $T \times L \times T \times ... \times T$. The first T is the return type, followed by the label, and all the parameter types. As an example, the Java function:

```
String toString(int i, int radix)
```

is modeled as

(`java.lang.String`, (**to**, **string**), `int`, `int`)

For convenience, we'll also define $ret(f)$, $label(f)$ and $params(f)$ to be the return type, label, and parameter types, respectively, of a function f.

4.4 Function Tree

Recall that the purpose of defining types, labels and functions is to model programming language expressions. We model a programming language expression as a function tree. Each node in the tree is associated with a function from F, and obeys certain type constraints.

In particular, a node is a tuple consisting of an element from F followed by some number of child nodes. For a node n, we'll define $func(n)$ to be the function, and

$children(n)$ to be the list of child nodes. The number of children in a node must be equal to the number of parameter types of the function, i.e., $|children(n)| = |params(func(n))|$. The return types from the children must also fit into the parameters, i.e., $\forall_i ret(func(children(n)_i)) \in sub(params(func(n))_i)$.

Note that function trees are only used internally by the algorithms. In the end, the system will render the function tree as a syntactically-correct and type-correct expression in the underlying programming language.

4.5 Java Mapping

To give a sense for how a programming language would map to this model, here are the particulars for mapping various Java elements to T, L and F. Most of these mappings are natural and straightforward.

4.5.1 Classes

A class or interface c is modeled as a type in T, using its fully qualified name (e.g. `java.lang.String`). Any class that is assignment compatible with c is added to $sub(c)$, including any classes that extend or implement c.

4.5.2 Primitive Types

Because of automatic boxing and unboxing in Java 1.5, primitive types like `int`, and `char` are modeled as being the same as their object equivalents `java.lang.Integer` and `java.lang.Character`.

4.5.3 Methods

Methods are modeled as functions that take their receiver object as the first parameter. For instance, the method:

```
    public Object get(int index)
```

of `java.util.Vector` is modeled as:

(`java.lang.Object`, (**get**), `java.util.Vector`, `int`)

4.5.4 Fields

Fields become functions that return the type of the field, and take their object as a parameter. For instance, the field:

```
public int x
```

of `java.awt.Point` is modeled as:

(`int`, (**x**), `java.awt.Point`)

Note that assignment to fields is more difficult, and is discussed in the *Other Java Mappings* section below.

4.5.5 Local Variables

Local variables are modeled by functions that return the type of the variable and take no parameters, e.g., the local variable `int i` inside a `for`-loop is modeled as (`int`, (**i**)).

4.5.6 Constructors

Constructors are modeled as functions that return the type of object they construct. We use the keyword **new** and the name of the class as the function label, e.g., the constructor for `java.lang.Integer` that takes a primitive `int` as a parameter is represented by:

(`java.lang.Integer`, (**new**, **integer**), `int`)

4.5.7 Members

Member methods and fields are associated with an additional function, to support the Java syntax of accessing these members with an assumed `this` token. The new function doesn't require the object as the first parameter. For instance, if we are writing code inside `java.awt.Point`, we would create a function for the field `x` like this: (`int`, (**x**)). Note that we can model the keyword `this` with the additional function (`java.awt.Point`, (**this**)).

4.5.8 Statics

Static methods do not need a receiver object—it is optional. One strategy to support the optional argument is to use two functions. For instance

```
static double sin(double a)
```

in `java.lang.Math` is modeled by adding both:

(`double`, (**sin**), `java.lang.Math`, `double`), and

(`double`, (**math**, **sin**), `double`)

Note that in the second case, **math** is included in the function label. This is done since experienced Java programmers are used to including the type name when calling static methods. This makes a difference to the algorithms because of the way function labels are used as explanations for keywords in a query. In this case, including **math** in the label for `sin` helps the algorithms know that this function is a good explanation both keywords in a user's query. Otherwise, the algorithms may spend time trying to find an alternative explanation for the keyword **math**.

An alternate approach is to have two types, instead of two functions. For instance, the additional type `static:java.lang.Math` could be added to T, and then the function `sin` could have one version, namely

(`double`, (**sin**), `static:java.lang.Math`, `double`)

The last step would be to make `java.lang.Math` a subtype of `static:java.lang.Math`, and add a special constructor function for static types, like

(`static:java.lang.Math`, (**math**))

.

4.5.9 Generics

Generics are supported explicitly, i.e., the system creates a new type in T for each instantiation of a generic class or method. For instance, if the current source file contains a reference to both `Vector<String>` and `Vector<Integer>`, then both of these types are added to T. The system also adds all the methods for `Vector<String>` separately from the methods for `Vector<Integer>`. For example, the `get` method would produce both

(`String`, (**get**), `Vector<String>`, `int`), and

(`Integer`, (**get**), `Vector<Integer>`, `int`)

The motivation behind this approach is to keep the model simple, and programming language agnostic. In practice, it does not explode the type system too much, since relatively few separate instantiations are visible from any particular scope.

One limitation of this approach is that it does not support creating new instances of generic types not already present in the source code. For instance, if a source file did not contain any references to the type `Vector<Integer>`, then a user could not type **new vector integer**, and expect to get `new Vector<Integer>()`.

An alternate approach is to simply erase generics, and treat them as Java 1.4 style objects, e.g., treat `Vector<String>` as just `Vector`. The downside is that the search problem becomes less constrained. On the other hand, T and F are smaller, making the algorithms faster.

4.5.10 Other Java Mappings

I have experimented with additional Java mappings, though these are not implemented in the algorithms, so there is no formal evaluation of them. However, literals are implemented in the Reference and Koala algorithms.

For some of these, it seems that the most natural implementation may involve complicating the type system. For instance, adding the ability to specify constraints between parameter types, like "the return type must be the same as the first parameter."

Literals:

Numeric literals can be added by expanding the notion of a function name to include regular expressions. For instance, integer literals could become (`int`, [-+]?[0-9]+).

String literals can also be represented as regular expressions, which would require the string to begin and end with a quote character. Ideally, we want to expand the notion of strings to not require quotes. Both Chickenfoot and Koala support this. Koala's approach is to deal with unquoted strings as a post-processing step, and this may be the best approach for Java as well.

Operators:

Operators can map to functions in the natural manner, but multiple versions of them are required to support all the primitive types that use them; e.g. we require (`int`, +, `int`, `int`) separate from (`long`, +, `long`, `long`). It might seem like there could just be (`double`, +, `double`, `double`), since all the other numeric primitives are subtypes of `double`. However, this wouldn't allow two numbers to be added, with the result passed to a function that requires an `int`.

Assignment:

Assignment gets a little more complicated. Considering wanting to allow the assignment `x = y`, where `x` is an `int` and `y` is a `short`. One solution is to add (`int`, =,

`int`, `int`). Unfortunately, this doesn't prevent users from passing subtypes of `int` to the left-hand side of the assignment, so this wouldn't prevent `y = x`. It also wouldn't prevent `5 = x`.

One approach I have tried is having the system add a special set-function for each variable. In this example, the system would add (`int`, `x =`, `int`). Note that the function name includes the `=`. This seems to work, but it does require adding lots of new functions.

A cleaner solution might involve adding more types; in particular, an "assignable" or "reference" type. So the variable `int x` would be accessible with the function (`ref:int`, `x`). There would also be a function for integer assignment: (`int`, =, `ref:int`, `int`). Finally, it would be necessary to make `ref:int` a subtype of `int`, so that users could still use `x` on the right-hand-side of an assignment. This technique still requires an assignment function for each type, but not for each variable.

Array Indexing:

Array indexing (e.g. `a[i]`) is possible with functions of the form (`String`, [], `String[]`, `int`). This technique requires adding a function like this for each type of array that we see. A potentially cleaner alternative would involve adding the function (t, [], t`[]`, `int`), along with the machinery in the algorithm to constrain the t's to be equal.

Other:

I have thought about how to add constructs like control flow and class declarations. One paradigm that seems interesting involves supporting line-by-line completions of these constructs. For instance, the user might enter the keywords **if x == y**, and the system might suggest `if (x == y) {`, or create a template of an `if` statement with `x == y` filled in. The function for `if` would then look like (`void`, **if**, `boolean`). Note that this function is not a function in the traditional sense of something the computer can execute; rather, it is a function that generates code for the user to work with. This is fine for this system since it is a code completer, and not an actual interpreter.

4.5.11 Other Language Mappings

Although most of the mappings in any language would follow the Java mapping pretty closely, there are some other issues to consider. For instance, Javascript is a dynamically typed language, and the function definitions in Javascript do not specify the type. Hence, when mapping Javascript to this model, a human must manually specify types which do not exist in the Javascript source. This is fine for small domains like Chickenfoot, though it is left to future work to deal with this in general. Another reason to leave this for future work is that other research efforts may mitigate the problem by recording these types based on dynamic runs of a script, or static analysis.

Chapter 5

Algorithms

Now that we have a formal model of the domain, we can provide a formal statement of the problem that our algorithms must solve.

The input to an algorithm consists of a model M, a keyword query, and a desired return type. The desired return type is made as specific as possible, e.g., when doing code completion, the system would try to determine the valid types given the source code around the keyword query. If any type is possible, then $\top$ is supplied as the desired return type. In the web prototype, `void` is used as the desired return type, since the commands are meant to *do* something, not return a value.

The output is a valid function tree, or possibly more than one. The root of the tree must return the desired type, and the tree should be a good match for the keywords according to some metric.

Choosing a good similarity metric between a function tree and a keyword query is the real challenge. The metric should match human intuition, and it should be easy to evaluate algorithmically.

The basic metric used in this thesis is based on the simple idea that each input keyword is worth 1 point, and a function tree earns that point if it *explains* the keyword by matching it with a keyword in the label of one of the functions in the tree. Note that the details of the metric vary somewhat in the algorithms presented below, and depend on their implementations and heuristics.

This chapter describes four algorithms for solving this problem. The Reference algorithm may be viewed as an ideal implementation of the keyword programming user interface—whereas the three subsequent algorithms fall short of this goal, and have various tradeoffs in terms of speed, accuracy, inference and domain specificity.

The Koala algorithm is much faster, but restricted to the Web domain. The Keyword Tree algorithm is faster, using a genetic algorithm to explore possible ways to arrange keywords into trees, but it cannot infer functions. The Bottom-up algorithm is a bottom up dynamic programming algorithm that keeps track of the best functions to achieve different types at each level of the tree; it is very fast, and can infer functions, but it cannot benefit from phrase structure information.

5.1 Reference Algorithm

The reference algorithm needs to convert a string of keywords into a likely function tree. It is described in two steps.

5.1.1 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. Tokens are further subdivided on word boundaries using several techniques. First, the presence of a lower-case letter followed by an upper-case letter is assumed to mark a word boundary. For instance, **LeftMargin** is divided between the **t** and the **M**. Second, common compound expressions are detected and split. For instance, **login** is split into **log in**. Note that we apply this same procedure when generating the keywords for function labels in the API.

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

The system uses spell correction to deal with this problem. The spelling dictionary includes all the keywords in L, but the system also 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 it is generating the label for this function. Now when the user enters **leftmargin**, the spell checker corrects it to **LeftMargin**, which is then split into **Left Margin**.

After spelling correction and word subdivision, tokens are converted to all lowercase, and then passed through a common stemming algorithm [26].

5.1.2 Step 2: Recursive Algorithm

The input to the recursive algorithm is a model M, a token sequence and a desired return type. The result is a valid function tree derived from the sequence that returns the desired type. This algorithm is called initially with the entire input sequence, and the desired return type of the whole expression. Recursive calls will take smaller parts of the input sequence, and return types that fit into other parts of the function tree being built.

The algorithm begins by considering every function in F 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. 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. This is currently handled by not inferring functions after a certain depth in the recursion.

5.1.3 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 (0.3 in the Web prototype). Also, inferred functions subtract some explanatory power, depending on how common they are. Inferring common functions costs less than inferring uncommon functions; for instance, the `click` function in the web prototype only subtracts 0.1. Currently, these values are hard coded, but in the future they should reflect an *a priori* probability from a corpus of usage data.

5.1.4 Speed

This algorithm uses brute force, and has a poor running time complexity. Suppose the user's input has n tokens, and every substring of tokens matches f functions in F, each taking a arguments. The first call to the recursive algorithm must try every way to divide the n tokens into $a+1$ substrings in any order (one for each argument plus the function name itself), which is $\binom{n-1}{a}(a+1)! = O(an^a)$. And since each substring matches f functions, this gives $O(fan^a)$ possibilities for each recursive call. Since the function tree for n tokens can have at most n nodes (ignoring function inference), the total search time would be $O((fan^a)^n)$.

This running time is manageable when f is small (tokens match few functions), a is small (most functions take few arguments), and n is small (users use few tokens). The evaluations suggest that this algorithm is sufficient for small command sets (under 20 commands) where the user is unlikely to enter large expressions (over 8 keywords).

5.2 Koala Algorithm

The algorithm for the Koala system that I worked on at IBM is perhaps the most different from the rest of the algorithms presented here. The big difference is that rather than trying to build a function tree out of the keywords, the system instead tries to enumerate all the possible function trees given M. This is possible when M has a sufficiently small set of functions, and the type constraints inhibit deep recursive trees. This is the case in Koala, since the functions are based on the HTML objects available in the current webpage, and each object has a small set of possible functions (e.g. a button can only be clicked).

The algorithms is described here in three basic steps, using the example slop **type Danny into the first name field** on a simple web form (see Figure 5-1).

First, the interpreter enumerates all the possible function trees given M. This is equivalent to enumerating all the actions associated with various HTML objects in the document, such as links, buttons, text boxes, combo-box options, check boxes, and radio buttons, since all the action functions (i.e. functions returning `void`) in M

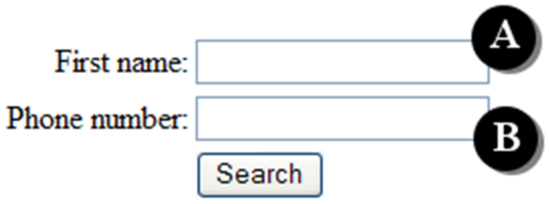


Figure 5-1: A simple web form.

require an HTML object as a parameter. For each of these objects, the interpreter effectively 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 a web form (A in Figure 5-1) 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— these come from the label of the function returning this object in F, namely `findTextbox` where `field` is a synonym for `textbox`. Finally, the interpreter would include verbs commonly associated with text fields such as **enter**, **type**, and **write**. Again, these come from the label of the `enter` function in F, and its synonyms.

Next, the interpreter searches for the function tree that matches the greatest number of keywords with the slop. 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, a good partition is: **type into the first name field** (with 4 keyword matches), and **Danny** (which gets points for being followed by the preposition **into** in the original slop). Note that this technique may not scale well to function trees which require multiple string arguments.

5.3 Keyword Tree Algorithm

In the Keyword Tree algorithm, a genome is defined that arranges keywords into a parse tree. The parse tree is then scored based on the function trees than can be created from it. Finally, a genetic algorithm is used to search over the space of possible parse trees for the one that can produce the highest scoring function tree.

5.3.1 Genome

Although genetic algorithms can be applied to trees directly, this can be complicated. This algorithm uses a simple linear genome that specifies how to arrange the keywords into a parse tree. This allows the use of standard cross-over to merge genes.

Consider that there are n keywords, $k_1, k_2, ..., k_n$. To start, assume that each keyword is a singleton tree. The genome consists of $n-1$ components $c_1, c_2, ..., c_{n-1}$. Each component can be thought of as existing between two keywords. In particular, define $left(c_i) = k_i$ and $right(c_i) = k_{i+1}$. Each component c is a 3-tuple: $(rule, order, index)$ where:

- $rule(c)$ is an element of $\{\leftrightarrow, \nearrow, \nwarrow\}$ that defines the relationship between $left(c)$ and $right(c)$. The $\leftrightarrow$ has the meaning: make $left(c)$ and $right(c)$ part of the same node. If $root(a)$ is defined as the root node of the tree containing a, then the $\nearrow$ has the meaning: make $root(left(c))$ a child of $root(right(c))$ (and $\nwarrow$ has the opposite meaning).

- $order(c)$ is a real number between 0 and 1 that determines the order in which to apply the rules. These are generated randomly, and it is assumed that each $order$ value will be unique.

- $index(c)$ is also a number in the range 1 to $n-1$ that specifies where to insert nodes when one node is made to be the child of another node. If a node has n_c children, it is inserted at position $index \mod (n_c + 1)$.

Figure 5-2 is psuedocode for constructing the parse tree. Let $root(a)$ denote the root node of the tree containing a. Let $combine(a, b)$ have the side effect of combining

```
$C \leftarrow \{c_1, c_2, ..., c_{n-1}\}$
for each $c \in C$
    do  if $rule(c) = \leftrightarrow$
            then  $combine(root(left(c)), root(right(c)))$
                  $C \leftarrow C - \{c\}$
while $C \neq \emptyset$
    do  /* get component with smallest order */
        $c \in \{a \mid a \in C$ and $\forall_i\ order(a) \leq order(a_i)\}$
        /* apply the rule for c */
        if $rule(c) = \nearrow$
            then  $parent \leftarrow root(right(c))$
                  $child \leftarrow root(left(c))$
        else if $rule(c) = \nwarrow$
            then  $parent \leftarrow root(left(c))$
                  $child \leftarrow root(right(c))$
        $i \leftarrow index(c) \mod (|children(parent)| + 1)$
        $insert(parent, child, i)$
        $C \leftarrow C - \{c\}$
```

Figure 5-2: Pseudocode for constructing a parse tree.

the nodes a and b into a single node. Let $insert(a, b, i)$ have the side effect of inserting b into a at position i (using 0-based indexing).

Here is an example to illustrate. Consider the the following keyword query, based on the Java expression `boxes.addBox(b)`:

boxes add box b

There are 4 keywords, so there are 3 components in the genome. Let's say the components are: ($\nearrow$, 0.3, 2), ($\leftrightarrow$, 0.2, 3), and ($\nwarrow$, 0.1, 1). These can be visualized as sitting between the keywords:

boxes $_{(\nearrow,\ 0.3,\ 2)}$ **add** $_{(\leftrightarrow,\ 0.2,\ 3)}$ **box** $_{(\nwarrow,\ 0.1,\ 1)}$ **b**

In the first pass, the components with a *rule* of $\leftrightarrow$ are used to combine neighboring keywords into single nodes. There is such a component between **add** and **box**, so they are combined into a single node:

boxes $_{(\nearrow,\ 0.3,\ 2)}$ **add-box** $_{(\nwarrow,\ 0.1,\ 1)}$ **b**

Next, the remaining component with the lowest *order* is selected, which is ($\nwarrow$, 0.1, 1) between **add-box** and **b**. The *rule* is $\nwarrow$, so **b** is made to be a child of **add-box**. In this case there is only one place to insert it, so the *index* is ignored. Let **b(a)** denote making **a** the child of **b**, giving us:

boxes $_{(\nearrow,\ 0.3,\ 2)}$ **add-box(b)**

The final component says to make **boxes** a child of **add-box**. Since **add-box** already has a child **b**, the *index* of 2 is used to determine where to insert the new child. The *index* is taken modulus one more than the number of children already present, which is written as $2 \mod (1+1)$. This evaluates to 0, so **boxes** is inserted as the new first child of **add-box**, yielding:

add-box(boxes, b)

Note that this parse tree has the same structure as the abstract syntax tree (AST) for the original Java expression `boxes.addBox(b)`, which is good. This demonstrates that the genome is expressive enough to build this tree from the keywords. Next a sketch a proof is presented showing that in general, the genome is expressive enough to build any valid parse tree from the keywords.

Proof Sketch: The idea of the proof is to construct the component values given a desired tree. First note that every edge in the tree represents a component. (There are also components between the keywords in a single node, but these are trivial to deal with; just set the *rule* for each of these components to $\leftrightarrow$). Now take any one of the edges connected to the root, and find the corresponding component c. Then set $order(c)$ to the largest unused *order* value, and set the component $index(c)$ to the 0-based index of the child connected to this edge. Then set *rule* to $\nearrow$ if $root(right(c))$ is the root node, and set *rule* to $\nwarrow$ if $root(left(c))$ is the root node.

Next, remove this edge from the tree and repeat the process (select any edge from any of the remaining roots in the resulting forest of trees). It is simple to show that applying these construction rules in the reverse order will build up the tree exactly the way it was torn down. ■

5.3.2 Fitness Function

A parse tree defines a tree structure, but it does not specify which function should be used at each node. Consider the parse tree from the previous example:

add-box(boxes, b)

This has the same structure as these function trees:

A. addBox(boxes(), b()),

B. removeBox(boxes(), b()), and

C. synchronizedAddBox(boxes(), b())

However, it would be nice to say that it corresponds more to **A** than to **B**, because the root node of **A** matches more keywords with the root node of the parse tree. Now **A** and **C** both match the same number of keywords, but it would be nice to say that **C** is worse because it includes an unmatched keyword **synchronized**.

This notion is formalized in the way a function f is scored with a node n. First, let $key(n)$ represent the list of keywords in n. The score equals the number of shared keywords between $key(n)$ and $label(f)$, minus a small amount (0.01) for each keyword that is present in one, but not the other. Note that $key(n)$ and $label(f)$ are lists, and might contain repeated keywords (e.g. the static function `BorderFactory.createEmptyBorder` repeats the keyword **border**). If a keyword k appears x times in $key(n)$ and y times in $label(f)$, then $\min(x, y)$ points are awarded for this keyword, and $0.01 * (\max(x, y) - \min(x, y))$ points are subtracted.

To determine the score for the entire parse tree, define $score(n)$ as the score for the parse tree rooted at n. Also define $score(n, t)$ as the score of the best function tree derived from the parse tree rooted at n that returns type t. It is often true that $score(n) = \max_{t \in T} score(n, t)$, but not always, since it is not always possible to build a valid function tree that corresponds with the parse tree.

The calculation for $score(n)$ and $score(n, t)$ is done recursively. To process node n, the algorithm first processes the children of n. Next, for each function f, it calculates a score s. It will then update $score(n)$ and $score(n, ret(f))$ based on s.

The score s is initialized with the score calculated between $label(f)$ and $key(n)$, as described above. To speed up the algorithm, it is possible to skip the function if $s \leq 0$. Next, a point is added to s for each genome component with rule $\leftrightarrow$ that went into forming n, since these genome components are explained as creating n. Components can also be explained as combining nodes together, which is discussed next.

The algorithm then adds points to s associated with each child n' that is associated with a parameter type p from $params(f)$. If $score(n', t)$ is defined for some $t \in sub(p)$, then the algorithm adds $\max_{t \in sub(p)} score(n', t)$ to s, in addition to a point for the genome component that attached this child.

If the child doesn't return any types that can be used, or is not associated with a parameter type (because there are more children than parameters), then the algorithm doesn't add a point for the genome component that attached this child. However, it does add $score(n')$ to s, since it may be possible to use this subtree when the genes get mutated or merged with other genes.

Finally, $score(n)$ is updated to $\max(s, score(n))$. Also, $score(n, ret(f))$ is updated to $\max(s, score(n, ret(f)))$, but only if children were found returning the proper types for each parameter of f.

5.3.3 Genetic Algorithm

The algorithm starts with an initial population of 100 genomes, where each genome is initialized with random values for each component. Each subsequent generation has only 10 genomes. The algorithm is run for 100 generations. Each generation is created from the previous generation. The best genome from the previous generation is copied directly into the new one, while every other genome is created from two parents selected randomly in the following manner: the previous generation is put in order of best fitness score, and the index of each parent is chosen using a normal distribution in the following equation: $|\lfloor N(0, 3^2) \rfloor| \mod 10$.

New genomes are created using cross-over and mutation. One cross-over point is chosen. Components before the cross-over point are copied from one parent, and components after the cross-over point are copied from the other. Each element of each component has a 20% chance of mutating to a random value.

5.3.4 Extraction

After running the genetic algorithm, the result is a high-scoring parse tree. The next step is to extract the best function tree from it. This is done by adding additional bookkeeping to the scoring algorithm to keep track of which function achieves $score(n, t)$ at node n for a given type t.

Given this information, a simple recursive algorithm is used to build the function tree. It looks at the root node to find the best function that returns the desired type (or any subtype). Then this process is repeated recursively to find the best function that achieves the desired parameter type from each child.

5.3.5 Limitations

This algorithm will only consider trees that obey phrase structure, i.e., each subtree is constrained to be a contiguous group of keywords. This means that if a user enters **my print message**, when they mean `print(myMessage)`, the system will not consider the tree **print(myMessage)**, since this tree groups **my** and **message** into a subtree, whereas they are not contiguous in the input. However, the algorithm may still make the proper suggestion if it cannot find a use for the isolated keyword **my**. For instance, it could find the tree **myPrint(message)**, which may still result in the function tree `print(myMessage)` (since **myPrint** is a pretty good match for `print`, and **message** for `myMessage`).

A bigger limitation is that this algorithm does not support function inference. This means that a user could not type **print myMessage**, and expect the system to generate `System.out.println(myMessage)`, since the system cannot infer the field `System.out`. The next algorithm overcomes both of these limitations.

5.4 Bottom-Up Algorithm

This algorithm can be thought of as a dynamic program which fills out a table of the form $func(t, i)$, which says which function to use to achieve type t, if the function tree can be at most height i (where a tree with a single node is defined to have height 1). It also records the expected score of the resulting function tree.

Calculating $func(t, 1)$ for all $t \in T$ is relatively easy. The algorithm only considers functions that take no parameters, since the tree height is bounded by 1. Then for each such function f, the algorithm gives a score based on its match to the user's keywords, and it associates this score with $ret(f)$. Then for each $t \in T$, $func(t, 1)$ is updated with the best score associated with any subtype of t.

Instead of a scalar value for the score, an *explanation vector* is used. This is explained next, before describing the next iteration of the dynamic program.

5.4.1 Explanation Vector

The idea of the explanation vector is to encode how well the input keywords have been explained. If there are n keywords $k_1, k_2, ..., k_n$, then the explanation vector has $n + 1$ elements $e_0, e_1, e_2, ..., e_n$. Each element e_i represents how well the keyword k_i has been explained on a scale of 0 to 1; except e_0, which represents explanatory power not associated with any particular keyword. When two explanation vectors are added together, the algorithm ensures that the resulting elements $e_1, e_2, ..., e_n$ are capped at 1, since the most that a particular keyword can be explained is 1.

Before anything else is done, the algorithm calculates an explanation vector $expl(f)$ for each function $f \in F$. In the common case, it sets e_i to 1 if $label(f)$ contains k_i. For instance, if the input is:

is boxes empty

and the function f is (`boolean`, (**is**, **empty**), `List`), then $expl(f)$ would be:

$(e_0, 1_{\textbf{is}}, 0_{\textbf{boxes}}, 1_{\textbf{empty}})$

In the Keyword Tree algorithm, unmatched keywords were penalized. This algorithm does the same thing by setting e_0 to $-0.01x$, where x is the number of words appearing in either the input or $label(f)$, but not both. In this case, it sets e_0 to -0.01, since the word **boxes** does not appear in $label(f)$.

Now consider the input:

node parent remove node

where `node` is a local variable modeled with the function (`TreeNode`, (**node**)). Since **node** appears twice in the input, the explanation of the word **node** is distributed between them:

$(e_0, 0.5_{\mathbf{node}}, 0_{\mathbf{parent}}, 0_{\mathbf{remove}}, 0.5_{\mathbf{node}})$

In general, the algorithm sets $e_i = \max(\frac{x}{y}, 1)$, where x is the number of times k_i appears in $label(f)$, and y is the number of times k_i appears in the input.

In this case the algorithm sets e_0 to -0.03, since there are three words that appear in the input, but not in the function label (it includes one of the **node** keywords in this count, since it only appears once in the label).

5.4.2 Next Iteration

In the next iteration, the goal is to compute $func(t, i)$ for all $t \in T$. The basic idea is to consider each function f, and calculate an explanation vector by summing the explanation vector for f itself, plus the explanation vector for each parameter type p found in $func(p, i - 1)$.

This can be done, but there is a problem. It is no longer guaranteed that we have the optimal explanation vector possible for this function at this height. Consider the following input:

add x y

and assume that there are the functions:

(`int`, (**add**), `int`, `int`),

(`int`, (**x**)), and

(`int`, (**y**)),

The data stored at $func(\texttt{int}, 1)$, is likely to be either (`int`, (**x**)), or (`int`, (**y**)). Assume it is (`int`, (**x**)). Now consider what happens in the next iteration when the algorithm processes the function (`int`, (**add**), `int`, `int`). Now the explanation vector (-0.02, $1_{\mathbf{add}}$, $0_{\mathbf{x}}$, $0_{\mathbf{y}}$), is added to the explanation vector found in $func(\texttt{int}, 1)$, which is (-0.02, $0_{\mathbf{add}}$, $1_{\mathbf{x}}$, $0_{\mathbf{y}}$). The results is (-0.04, $1_{\mathbf{add}}$, $1_{\mathbf{x}}$, $0_{\mathbf{y}}$).

Next, the algorithm adds the explanation vector for the second parameter, which is also type `int`. Again, it looks in $func(\texttt{int}, 1)$ and finds (-0.02, $0_{\mathbf{add}}$, $1_{\mathbf{x}}$, $0_{\mathbf{y}}$). When this is added, the result is (-0.06, $1_{\mathbf{add}}$, $1_{\mathbf{x}}$, $0_{\mathbf{y}}$), since the keyword components are capped at 1.

But what if the algorithm had found the explanation vector for (`int`, (**y**))? Then it could have gotten (-0.06, $1_{\mathbf{add}}$, $1_{\mathbf{x}}$, $1_{\mathbf{y}}$), which is better.

To get around this problem, the top r functions are stored at each $func(t, i)$, where r is an arbitrary constant. In my experiments, I chose $r = 3$, except in the case of $func(\texttt{java.lang.Object}, i)$, where I keep the top 5 (since many functions return this type).

Now when considering function f at height i, and the algorithm is adding explanation vectors for the parameters, it is greedy: it adds the explanation vector that increases the final vector the most, and then moves on to the next parameter. Note that if the parameter type is p, it considers all the explanation vectors in each $func(p, j)$ where $j < i$.

5.4.3 Extraction

After the dynamic program has been run to some arbitrary height h (in this case, $h = 2$), a function tree needs to be extracted.

A greedy recursive algorithm is used (see Figure 5-3) which takes the following parameters: a desired return type t, a maximum height h, and an Explanation Vector

```
procedure EXTRACT TREE($t, h, e$)
 for each $f \in func(t, i)$ where $i \leq h$
   do
     /* create tuple for function tree node */
     $n \leftarrow (f)$
     $e_n \leftarrow e + expl(f)$
     /* put most specific types first */
     $P \leftarrow$ SORT($params(f)$)
     for each $p \in P$
       do
         $n_p, e_p \leftarrow$ EXTRACT TREE($p, i-1, e_n$)
         /* add $n_p$ as a child of $n$ */
         $n \leftarrow append(n, n_p)$
     if $e_n > best_e$
       then
         $best_e \leftarrow e_n$
         $best_n \leftarrow n$
 return $(best_n, best_e)$
```

Figure 5-3: Pseudocode to extract a function tree.

e (representing what has been explained so far). The function returns a new function tree, and an Explanation Vector. Note that the parameters are sorted such that the most specific types appear first (t_1 is more specific than t_2 if $|sub(t_1)| < |sub(t_2)|$).

5.4.4 Running Time

This thesis does not provide a formal analysis of the running time for the Keyword Tree or Bottom-up algorithms, but they are both able to generate function trees in well under a second with thousands of functions in F, hundreds of types in T, and up to 6 keywords. With more than 6 keywords, the Keyword Tree algorithm starts to take longer than a second. The Bottom-up algorithm remains about 0.5 seconds even for large inputs (more 12 tokens). These figures are provided in more detail in the evaluation.

Chapter 6

Applications

This chapter describes several prototypes which were built in order to test various the different algorithms. The first prototype uses the reference algorithm in a web based system which lets users interact with a web browser using a command line interface. The second prototype applies the same algorithm to a command line interface to Microsoft Word. The third prototype is the Koala system I worked on at IBM, which also interacts with a web browser, but uses the a different algorithm. The fourth prototype is an Eclipse Plugin that tests the Keyword Tree and Bottom-up algorithms in the domain of general purpose Java programming.

6.1 Web Prototype

This prototype was build to test the hypothesis that a keyword 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 prototype takes the form of a command interface which users can use to perform common web browsing tasks.

The functions in the web prototype map to commands in Chickenfoot [5]. There are a total of 18 functions, and synonyms have been added, such that there is an average of 6.4 synonymous names for each function. Many functions share some of the same names. For instance, `make` is a name for the `pick` function and the `enter`

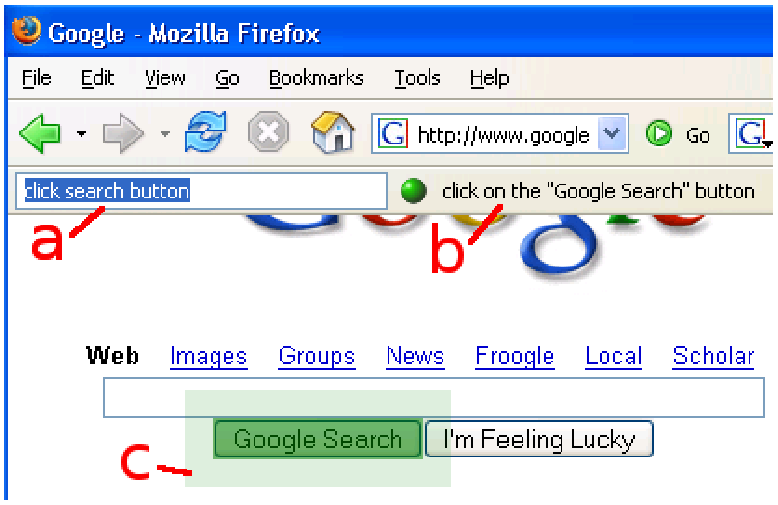

Google - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.google
Go
click search button
click on the "Google Search" button
a
b
Web Images Groups News Froogle Local Scholar
Google Search
I'm Feeling Lucky
c

Figure 6-1: a) command box, b) feedback bar, c) animated acknowledgment

function. The algorithm can resolve these ambiguities based on other queues, like argument types.

The GUI for this prototype consists of a textbox affording input, and an adjacent horizontal bar allocated for textual feedback. The system also generates an animated acknowledgment in the web page around the html object affected by a command (see Figure 6-1).

A plan for future work is to extend this system with a dropdown menu to present a list of likely interpretations for ambiguous commands (the current prototype makes an arbitrary choice in such cases).

6.2 Word Prototype

Part of the purpose of the Microsoft Word domain prototype was to explore what changes can be made to the Reference algorithm in order to scale it to larger domains. Chickenfoot is a fairly small domain, with less than 20 functions in F, whereas the Word API has over two thousand.

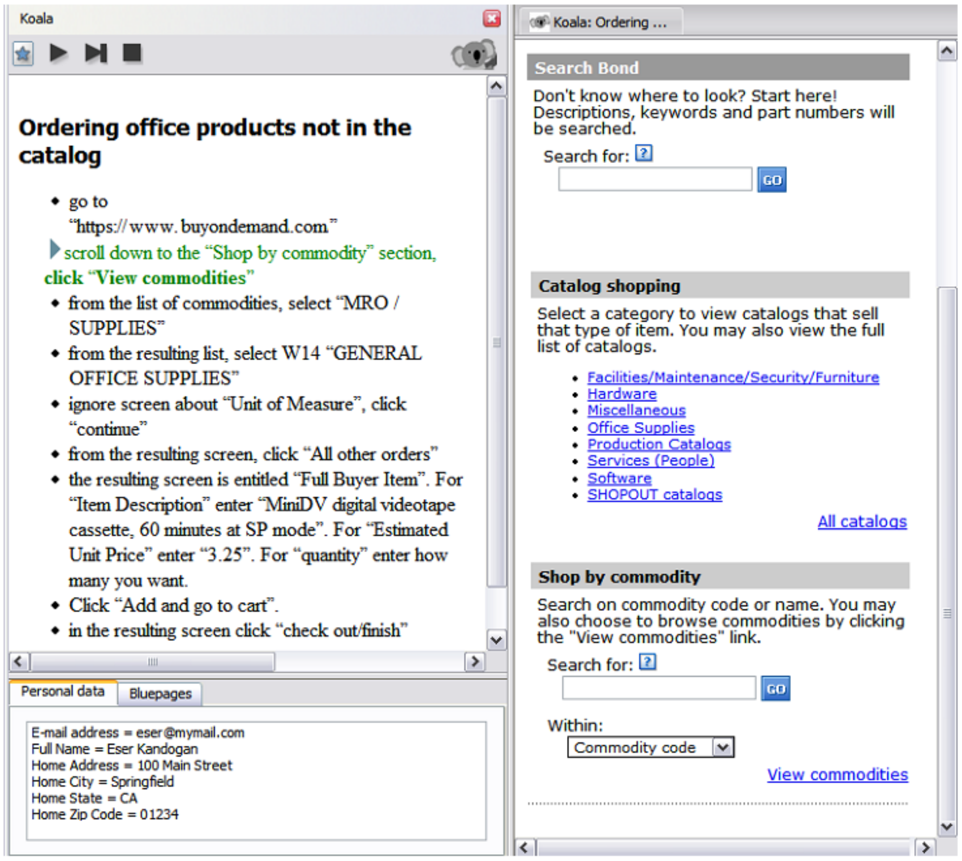
In order to expose all the functions to the system, Word's type libraries were mined, and its properties and methods were modeled as functions. No synonyms were added to any functions for this system.

Several changes were made to the algorithm to cope with the increased number of functions. The first change limited function inference to functions which required no additional parameters, since exploring these possibilities accounted for much of the search time. It was also noted that many useful properties in the system were descendants of `ActiveDocument` or `ThisApplication`, like `ActiveDocument.-PageSetup.TextColumns`. In order to support accessing these properties without supplying keywords for all the parent objects, the system models these properties as static methods.

The second modification was to use argument names as synonyms for function names. This helped in cases like the following: Consider the keyword query **columns 2**. The desired interpretation is `ActiveDocument.PageSetup.TextColumns.Set-Count(2)`, but this requires inferring the function `SetCount` since no words from this function appear in the original expression. However, note that `SetCount` has an argument named `NumColumns`, and the keyword **Columns** *does* appear in the expression. Hence, including `NumColumns` in the function label for `SetCount` makes the property accessible with this query.

The final modification is to search for functions with any return type, instead of just functions returning `void`. Then, if the return is not `void`, the system tries to fit the return value into some other function that does return ***void***. As an example, consider the expression **A4**, where the desired translation is `ActiveDocument.-PageSetup.PaperSize = wdPaperA4`. In this case, the system needs to infer the function `PaperSize`, but it gets around this searching for functions with any return type. In this example, the system returns the function `wdPaperA4`, which has the return type `WdPaperSize`. The system then searches for a function which takes something of type `WdPaperSize` as an argument. Remarkably enough, there is only one such function, namely the setter for `PaperSize`.

Despite these modifications, the Word prototype was relatively slow, as shown in the next chapter. However, exploring this potential helped shape the design of the Keyword Tree and Bottom-up algorithms.



Ordering office products not in the catalog

- go to "https://www.buyondemand.com"
- scroll down to the "Shop by commodity" section, click "View commodities"
- from the list of commodities, select "MRO / SUPPLIES"
- from the resulting list, select W14 "GENERAL OFFICE SUPPLIES"
- ignore screen about "Unit of Measure", click "continue"
- from the resulting screen, click "All other orders"
- the resulting screen is entitled "Full Buyer Item". For "Item Description" enter "MiniDV digital videotape cassette, 60 minutes at SP mode". For "Estimated Unit Price" enter "3.25". For "quantity" enter how many you want.
- Click "Add and go to cart".
- in the resulting screen click "check out/finish"

Personal data | Bluepages

E-mail address = eser@mymail.com
Full Name = Eser Kandogan
Home Address = 100 Main Street
Home City = Springfield
Home State = CA
Home Zip Code = 01234

Search Bond

Don't know where to look? Start here! Descriptions, keywords and part numbers will be searched.

Search for: ? GO

Catalog shopping

Select a category to view catalogs that sell that type of item. You may also view the full list of catalogs.

- Facilities/Maintenance/Security/Furniture
- Hardware
- Miscellaneous
- Office Supplies
- Production Catalogs
- Services (People)
- Software
- SHOPOUT catalogs

All catalogs

Shop by commodity

Search on commodity code or name. You may also choose to browse commodities by clicking the "View commodities" link.

Search for: ? GO

Within: Commodity code

View commodities

Figure 6-2: Ordering Office Products using Koala: Tina launches Koala, copies instructions from an email of a colleague, steps through the instructions, fixes steps manually, if necessary, to complete an order not in catalog.

6.3 Koala

I worked on the Koala system at IBM. The motivation behind this prototype was to explore the synergy of combining several programming paradigms, of which keyword programming was one. The other paradigms include programming by demonstration, and the use of a wiki as a form of end user source control.

This prototype experimented with additional user interface techniques to assist users with the keyword 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 Figure 6-2.

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 planning to do, and why?

The system shows the user what it is planning to do by placing a transparent green rectangle around the View commodities link, which is also scrolled into view.

The system addresses the question of "why" by letting the user know which words in their keyword query lead the interpreter to its selection. 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:

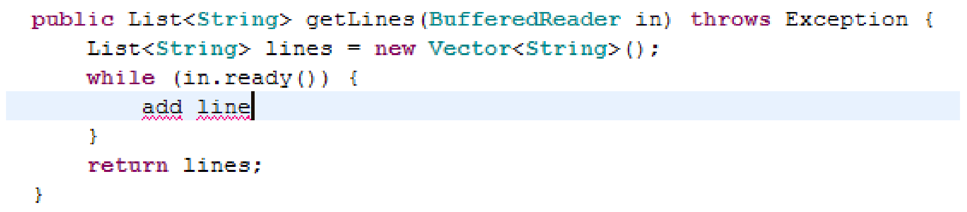
- click the "View commodities" link
- click the "View contracts" link
- click the "Skip to navigation" link

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, they 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.

6.4 Eclipse Plugin

The purpose of the Eclipse Plugin is to explore both the speed and usability of the Keyword Tree and Bottom-up algorithms in the domain of general purpose Java programming. The plugin integrates with Eclipse's Java editor, and provides a form of autocomplete for the user.

With the Eclipse Plugin installed, the user may enter keyword queries directly into their source code. The following figure shows the user entering the keywords **add line**:



```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        add line
    }
    return lines;
}
```

Next, the user presses *Ctrl-Space* to bring up Eclipse's autocomplete menu, and a hook is added that does the following:

1. It updates the model M with local variables in the current context. For instance, it sees the local variable `lines`, so it adds the function (`List<String>`, (**lines**)). The model is initialized in the background based on the classes named in the current source file. For instance, when the system sees the class name `List<String>`, it adds all the methods and fields associated with `List<String>`.

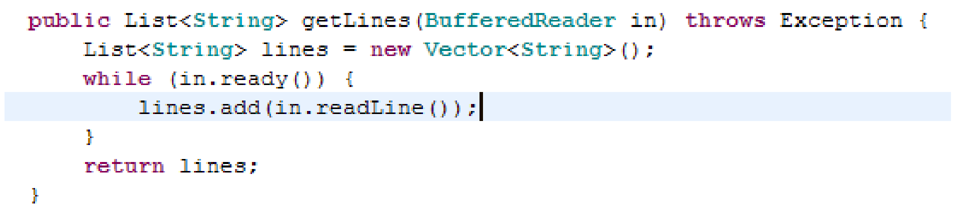
2. It figures out where the keyword query begins (it assumes that it ends at the cursor). It uses some heuristics to make this determination, including the nearest Java compilation error, which occurs on the keyword **add** in this example.

3. It tries to determine 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 keywords **in ready** into the condition for the `while` loop, then the system would expect a `boolean` return type.

The system now has all the information it needs to use one of the algorithms. The algorithm will suggest a valid function tree given the constraints. In this example, the Bottom-up algorithm finds the tree **add(lines, readLine(in))**. The system then renders this in Java syntax as:

```
lines.add(in.readLine())
```

Since it is the only completion available, Eclipse automatically replaces the keyword query in the source code:



```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        lines.add(in.readLine());
    }
    return lines;
}
```

If the user is not satisfied with the result, they can press *Ctrl-Z* to undo it. One plan for future work is to support multiple suggestions in the case of ambiguity; the real challenge is adding support for this to the algorithms themselves.

Chapter 7

Evaluation

This chapter presents tests that were run on the prototypes. The first test attempts to gauge how intuitive keyword programming is for end users. This test is also used to provide a feel for the scalability of the reference algorithm (which is not high). The second test explores the speed and accuracy of the more advanced algorithms on the larger domain of general purpose Java programming. The third test attempts to duplicate the results of the first test in the domain of Java.

7.1 Web Prototype User Study

This study has users attempt to use the web prototype to complete various tasks with no instructions about how to use write keyword queries.

7.1.1 Participants

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

The subjects were also all experienced web users, and could type reasonably well.

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

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

7.1.2 Setup

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

Instructions:

The instructions indicated that they should use only the command box (Figure 6-1a) to do each task, and not click or type directly into the web page. They were also informed that it was up to them to decide what to type into the command box. No suggestions were given about what to type (except for two users, see *Modifications to Study* below).

Tasks:

Each of the 36 tasks consisted of a red circle drawn on a screen shot of the web browser, indicating what to do (see Figure 7-1). For instance, if a task required the user to navigate to a URL, then a circle would be drawn around the location bar loaded with that URL. This was done in order to minimize external hints about how the user should communicate with the system.

Even still, the circled text itself acted as a hint for many tasks, especially for

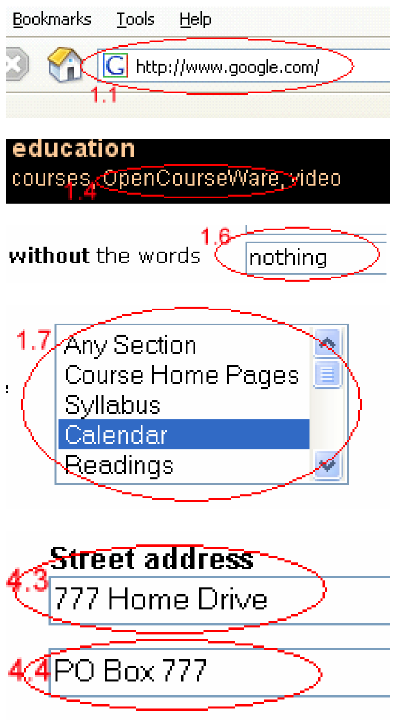


Figure 7-1: Examples of tasks in the user study. The user had to write a keyword command that would click or set the control circled in red. The red number is the task number.

following links and clicking buttons. In these cases, entering the text on the link or button itself was sufficient to click it. However, the subjects were not told that this was the case, and in fact, users often provided unnecessary words for these tasks (like the word **click**).

Some more difficult tasks were also included; these required words not present in the red circle to complete. However, every task could be completed with a single keyword query.

Modifications to Study:

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

I wondered what effect it would have if users were given a single initial suggestion of a command involving a verb. It seemed like it wouldn't overly contaminate the study to provide a hint for a task that every user had succeeded at (one of the easier tasks), while providing a small hint to satisfy my curiosity.

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

7.1.3 Results

The purpose of the user study on the web prototype was two-fold. First, to test the usability of the system, and second to test the speed of the algorithm.

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

The non-programmer group succeeded at 84% of the tasks, and the programmer group succeeded at 95% of the tasks. (This difference is statistically significant using a two-tailed t-test, with $p = 0.04$.) Each group averaged 1.7 attempts per task. Non-programmers completed 72% of the tasks on the first try, with only one command. The programmers achieved this for 77% of the tasks. If the system understood only JavaScript, and no instructions had been offered, the expected completion rate would be close to 0% for both groups, though this could vary depending on the quality of feedback (e.g. error messages) given to the user.

Figure 7-2 shows the average number of attempts made to accomplish each task (top chart), coupled with the average failure rate for each task (bottom chart). The tasks with labels are referenced in the *Discussion* below.

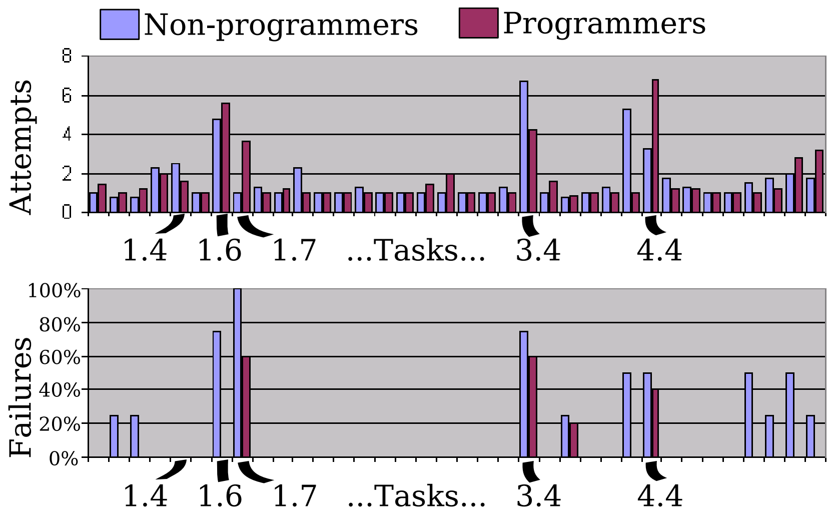


Figure 7-2: Average number of attempts made to accomplish each task (top chart), and average failure rate (bottom chart).

7.1.4 Discussion

Certain tasks exposed important issues with the system. The most was learned from the tasks with the highest failure rates. For reference, most of these tasks are shown in Figure 7-1.

Task 1.4:

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

Task 1.6:

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

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

It is worth saying something about one user in the programmer group with 22 attempts toward this task: most of these attempts were of the form ***next*** or ***up***, presumably in an attempt to move the attention of the computer to the textbox next to or above the most recent visual acknowledgment. This user appears to have mistaken the visual acknowledgments as describing the new focus of the system.

Task 1.7:

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

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

Task 3.4:

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

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

Task 4.4:

This was probably the hardest task in the study. It required the user to enter the address "PO Box 777" into the second street address field, shown at the bottom of Figure 7-1. People had difficulty identifying this field since it had no label of its own. Five subjects eventually succeeded, but 4 users did not. It is instructive to examine the potential reasons for these failures.

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

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

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

7.1.5 Speed

The translation algorithm for the web prototype was implemented in Java. The following running times provide a feel for the speed of this algorithm. Inputs from the user study were used to derive an average parse time of 44 milliseconds (on an AMD Athlon 4200+ processor). Figure 7-3 shows how the parse time varies for input sizes of different lengths. From this we can guess that the average-case running time is polynomial, but that it is reasonable for inputs up to 8 tokens long (taking less than 300 milliseconds on average). Note that this seemed adequate for tasks in the user study since 96% of the user inputs were 8 tokens or less.

The Word prototype was also implemented in Java, and it could translate many short expressions (4 words or less) in less than a second, however the running time increases significantly for longer expressions. It also depends on the words used. Including multiple words which appear in many commands can dramatically increase the translation time. For instance, the expression **left left** takes 4 seconds. Information from these tests was used to inform the design of the more advanced algorithms used in the Eclipse Plugin prototype.

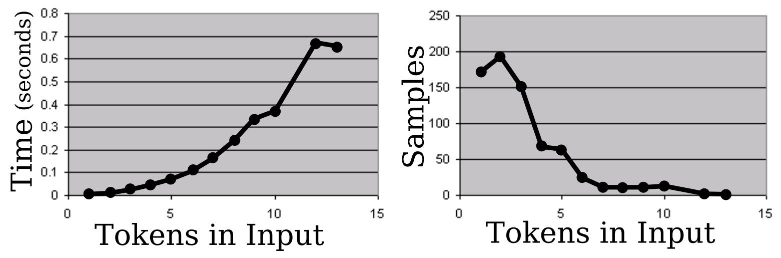


Figure 7-3: The left chart shows the time it took to parse inputs of various lengths, while the right chart shows how many samples we had of each length.

7.2 Eclipse Plugin on Artificial Corpus

The plugin and the algorithms were evaluated on an artificial corpus. The corpus consisted of a variety of open source Java projects. Artificial keyword queries were created by finding method calls and obfuscating them (removing punctuation and rearranging keywords). These keywords were then passed to the plugin, and the evaluation framework recorded whether it generated the original method call.

7.2.1 Projects

A lab mate selected 14 projects from popular open source web sites, including sourceforge.net, codehaus.org, and objectweb.org. Projects were selected based on popularity, and his ability to compile them using Eclipse. These projects were not selected with my tests in mind, so they should contain no bias toward the test. The projects include:

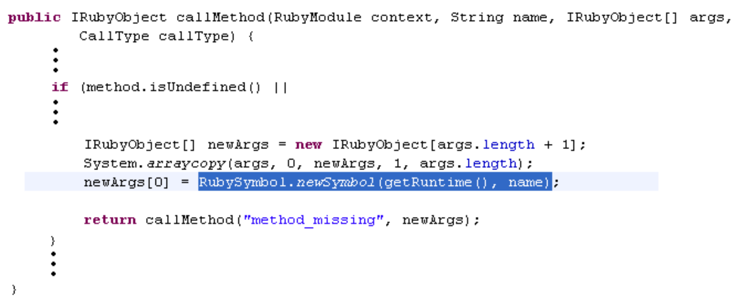
1. **Azureus**, an implementation of the BitTorrent protocol.
2. **Buddi**, a program to manage personal finances and budgets.
3. **CAROL**, a library for abstracting away different RMI (Remote Method Invocation) implementations.
4. **Dnsjava**, a Java implementation of the DNS protocol.
5. **Jakarta Commons Codec**, an implementation of common encoders and decoders.

Project	Class Files	LOC	Test Sites
Azureus	2277	339628	82006
Buddi	128	27503	7807
CAROL	138	18343	2478
Dnsjava	123	17485	2900
Jakarta CC	41	10082	1806
jEdit	435	124667	25875
jMemorize	95	14771	2604
Jmol	281	88098	44478
JRuby	427	72030	19198
Radeox	179	10076	1304
RSSOwl	201	71097	23685
Sphinx	268	67338	13217
TV-Browser	760	119518	29255
Zimbra	1373	256472	76954

Table 7.1: Project Statistics

6. **jEdit**, a configurable text editor for programmers.
7. **jMemorize**, a tool involving simulated flashcards to help memorize facts.
8. **Jmol**, a tool for viewing chemical structures in 3D.
9. **JRuby**, an implementation of the Ruby programming language in Java.
10. **Radeox**, an API for rendering wiki markup.
11. **RSSOwl**, a newsreader supporting RSS.
12. **Sphinx**, a speech recognition system.
13. **TV-Browser**, an extensible TV-guide program.
14. **Zimbra**, a set of tools involving instant messaging.

Table 7.1 shows how many class files and non-blank lines of code each project contains. The number of possible test sites are also reported, which is discussed in the next section.



```
public IRubyObject callMethod(RubyModule context, String name, IRubyObject[] args,
        CallType callType) {
    ⋮
    if (method.isUndefined() ||
    ⋮
        IRubyObject[] newArgs = new IRubyObject[args.length + 1];
        System.arraycopy(args, 0, newArgs, 1, args.length);
        newArgs[0] = RubySymbol.newSymbol(getRuntime(), name);

        return callMethod("method_missing", newArgs);
    }
    ⋮
}
```

Figure 7-4: Example Test Site

7.2.2 Tests

Each test is conducted on a method call, variable reference or constructor call. Only expressions of height 3 or less are considered, and it is made sure that they involve only the Java constructs supported by the model. For example, these include local variables and static fields, but do not include literals or casts. Expressions inside of inner classes are excluded since it simplifies the automated testing framework. Finally, test sites with only one keyword are discarded as trivial.

Figure 7-4 shows a valid test site highlighted in the JRuby project. This example is height 2, because the call to `getRuntime()` is nested within the call to `newSymbol()`. Note that nested expressions are counted as valid test sites as well, e.g., `getRuntime()` in this example would be counted as an additional test site.

To perform each test, the expression is obfuscated by removing punctuation, splitting camel-case identifiers, and rearranging keywords (while still maintaining phrase structure). This obfuscated code is then treated as a keyword query, which is passed on to the plugin. (Note that in these tests, the plugin is told explicitly where the keyword query starts and ends).

For example, the method call highlighted in Figure 7-4 is obfuscated to the following keyword query:

name runtime get symbol symbol ruby new

The plugin observes the location of this command in an assignment statement to `newArgs[0]`. From this, it detects the required return type:

```
org.jruby.runtime.builtin.IRubyObject
```

The plugin then passes the keyword query and this return type to one of the algorithms. In this example, it uses the Bottom-up algorithm, and the plugin returns the Java code:

```
RubySymbol.newSymbol(getRuntime(), name)
```

This string is compared with the original source code (ignoring white space), and since it matches exactly, the test is recorded as a success. Other information about the test is also recorded, including:

- **# Keywords:** the number of keywords in the keyword query.
- **time:** how many seconds the algorithm spent searching for a function tree. This does not include the time taken to construct the model. Tests were run in Eclipse 3.2 with Java 1.6 on an AMD Athlon X2 (Dual Core) 4200+ with 1.5GB RAM (the algorithms are single threaded).
- **$|T|$:** the number of types in the model constructed at this test site.
- **$|F|$:** the number of functions in the model constructed at this test site.

7.2.3 Results

The results presented here were obtained by randomly sampling 500 test sites from each project (except Zimbra, which is really composed of 3 projects, and 500 samples were taken from each of them). This provides 8000 test sites. For each test site, both algorithms were tested.

Table 7.2 shows how many samples there are for different keyword query lengths. Because there are not many samples for large lengths, all the samples of length 12 or more are grouped when plotting graphs against keyword length.

Figure 7-5 shows the accuracy of each algorithm given a number of keywords. Both algorithms have over 90% accuracy for inputs of 4 keywords or less. In most cases, the Keyword Tree algorithm is significantly more accurate than the Bottom-up

# Keywords	Samples
2	3330
3	1997
4	1045
5	634
6	397
7	206
8	167
9	86
10	54
11	38
$\geq$ 12	46
⋮	⋮
43	1

Table 7.2: Samples given # Keywords

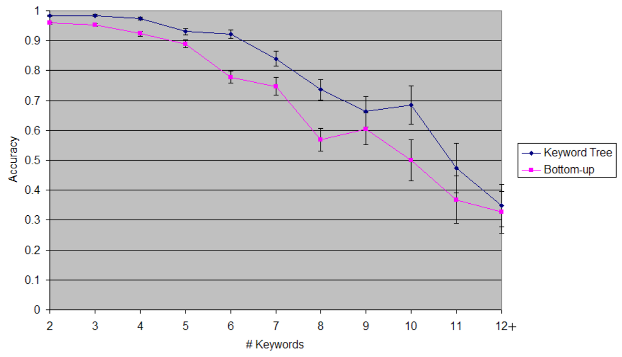


Figure 7-5: Accuracy given # Keywords

algorithm. This may be due in part to the fact that the Keyword Tree algorithm takes advantage of phrase structure, which is preserved by the obfuscation process.

Figure 7-6 shows how long each algorithm spent processing inputs of various lengths. The Bottom-up algorithm is slower for very small inputs, but grows at a slow rate, and remains under 500 milliseconds even for large inputs. The Keyword Tree algorithm doesn't scale as well to large inputs, taking more than 4 seconds for inputs 12 keywords long.

Another factor contributing to running time is the size of T and F in the model. Table 7-7 shows the average size of T and F for each project. The average size of T ranges from 100 to 800, while the average size of F ranges from 900 to 6000. Figure

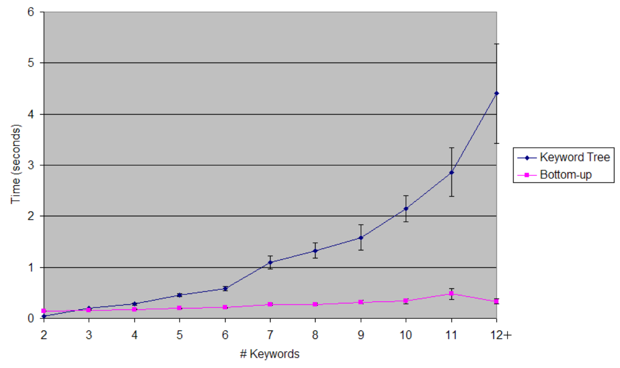


Figure 7-6: Time given # Keywords

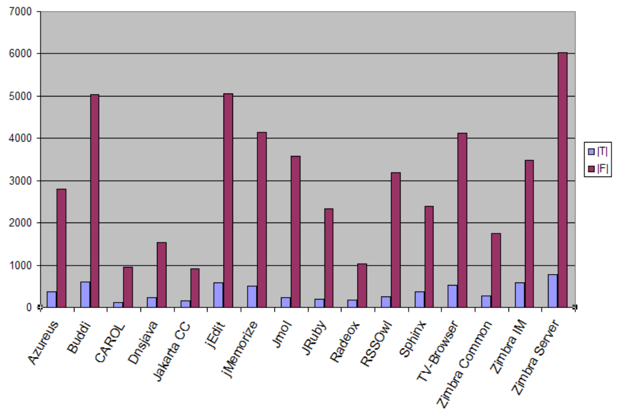


Figure 7-7: Size of T and F for different projects.

7-8 shows running time as a function of the size of F. Note that the Bottom-up algorithm takes about 1 second when F contains 14000 functions.

To test the inference capabilities of the Bottom-up algorithm, another set of tests was run on the same corpus. Only test sites with nested expressions were considered (i.e. the resulting function tree had at least two nodes). This ensured that when only one keyword was provided, the algorithm would definitely have to infer something.

Again, a random sample of 500 test sites was taken from each project. At each test site, the Bottom-up algorithm was run with the empty string as input. Next, the most unique keyword in the expression was chosen (according to the frequency counts in L), and the algorithm was run on this. Then the algorithm was run on the two most unique keywords, and the three most unique keywords etc... The left side of Figure 7-9 shows the number of keywords that were provided as input. The table

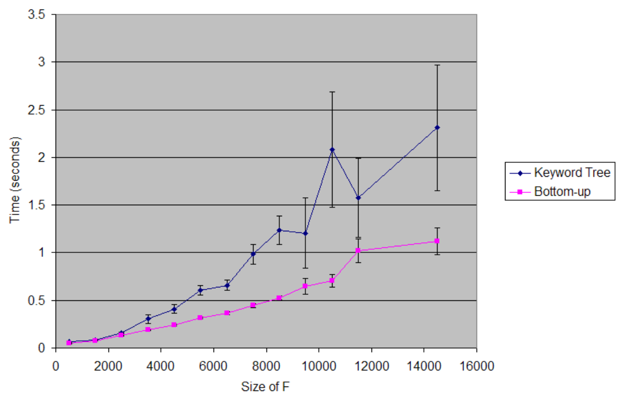
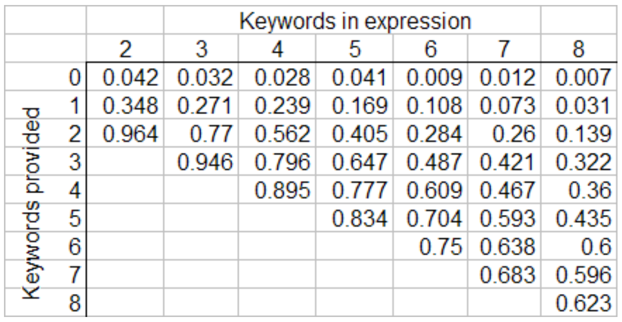


Figure 7-8: Time given size of F



	Keywords in expression						
Keywords provided	2	3	4	5	6	7	8
0	0.042	0.032	0.028	0.041	0.009	0.012	0.007
1	0.348	0.271	0.239	0.169	0.108	0.073	0.031
2	0.964	0.77	0.562	0.405	0.284	0.26	0.139
3		0.946	0.796	0.647	0.487	0.421	0.322
4			0.895	0.777	0.609	0.467	0.36
5				0.834	0.704	0.593	0.435
6					0.75	0.638	0.6
7						0.683	0.596
8							0.623

Figure 7-9: Accuracy of inference (Bottom-up algorithm)

shows the accuracy for different expression lengths (measured in keywords).

7.2.4 Discussion

These tests provide a rough upper and lower bound on the performance that can be expected from these algorithms. In the first test, the algorithms were provided with as much information as they are capable of using. In the second test (only conducted on the Bottom-up algorithm), the test began by providing the algorithm with as little information as possible (i.e. 0 keywords).

The results seem promising. The accuracies are certainly high for small inputs, and there are many accuracies over 50% for relatively challenging tasks, like inferring a 4 keyword expression from 2 keywords.

The hope is that this picture will improve even more if a list of results is provided, rather than the single best result; that is to say, when the algorithm fails to find the best result, hopefully the best result is still rated highly by the algorithm. In practice, a user would probably be satisfied to look through 5 results if they only had to type a couple keywords to obtain them.

7.3 Eclipse Plug-in User Study

The purpose of this study was to test the robustness of the algorithms on human generated inputs. Inputs were collected using a web-based survey targeted at experienced Java programmers.

7.3.1 Participants

Subjects were solicited from a public mailing list at a college campus, as well as a mailing list directed at the computer science department of the same college. Participants were told that they would be entered into drawing for $25, and one participant was awarded $25.

A total of 69 people participated in the study; however, users who didn't answer all the questions, or who provided garbage answers, like "dghdfghdf...", were removed from the data—though they were still eligible for the $25. This left 49 participants. Amongst these, the average age was 28.4, with a standard deviation of 11.3. The youngest user was 18, and the oldest user was 74. The vast majority of participants were male; only 3 were female, and 1 user declined to provide a gender. Also, 35 of the users were students, including 2 postdocs.

All of these users had been programming in Java for at least 2 years, except 2 people. One person had written a Java program for a class, as well as for a job. The other person had no Java experience at all, but had 20 years of general programming experience.

7.3.2 Setup

The survey consisted of a series of web forms that users could fill out from any web browser. Subjects were first asked to fill out a form consisting of demographic information, after which they were presented with a set of instructions, and a series of tasks.

Instructions:

Users were presented with instructions as shown in Figure 7-10. Users were meant to associate the small icons next to each text field with the large icons at the bottom of the page. Next to the large icons were printed instructions. The instructions asked the user to infer what the program did at the location of the text field in the code, and to write an expression describing the proper behavior. The instructions also prohibited users from looking online for answers.

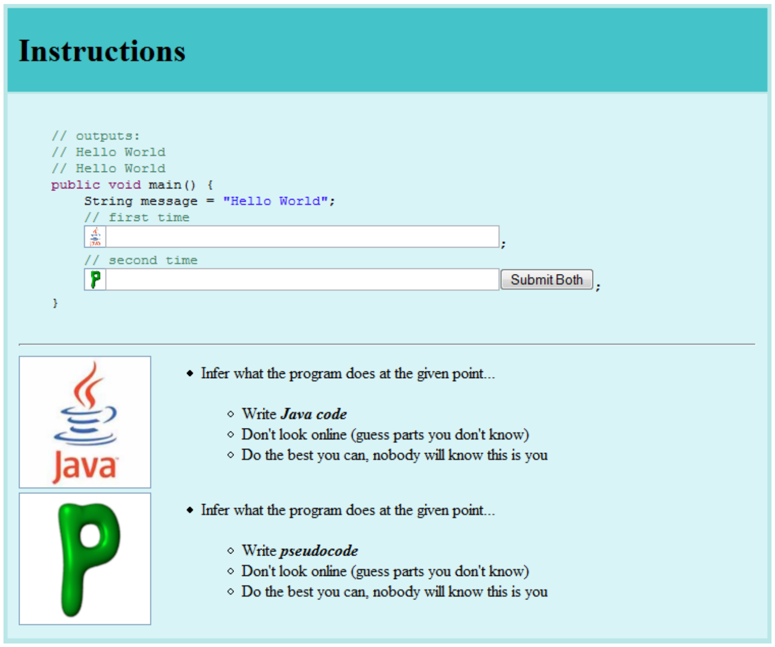
Each icon represented a different variant of the instructions. There were 3 variants: *Java*, *pseudocode*, and *keywords*. The Java and pseudocode variants simply asked the user to "write Java code" or "write pseudocode" respectively. The keywords variant said "Write **keywords** that suggest the proper code."

Each user would see 2 instruction variants in their instructions, though these could vary between users. The Java variant was always included, and either pseudocode or keywords was chosen randomly. The order that the variants were displayed was also random, but was made consistent with the order of the tasks, which are discussed next.

Tasks:

The survey consisted of 15 tasks. The same 15 tasks were used for each user, but the order was randomized. Each task asked for a single input, as opposed to the two inputs requested in the initial set of instructions.

The instructions for each task were determined based on the initial instructions shown to the user, e.g., if the first icon was keywords, then the first group of tasks


Instructions

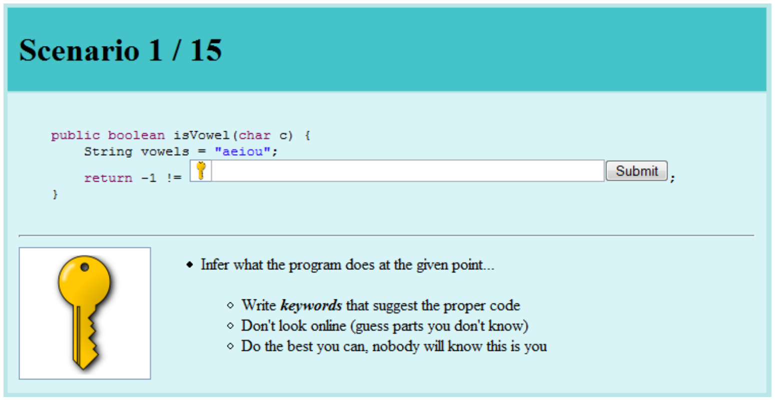
```
// outputs:
// Hello World
// Hello World
public void main() {
    String message = "Hello World";
    // first time
    [                              ];
    // second time
    [                              ] Submit Both ;
}
```

- Infer what the program does at the given point...
  - Write ***Java code***
  - Don't look online (guess parts you don't know)
  - Do the best you can, nobody will know this is you
- Infer what the program does at the given point...
  - Write ***pseudocode***
  - Don't look online (guess parts you don't know)
  - Do the best you can, nobody will know this is you

Figure 7-10: Sample set of instructions for participants in web-based study. Note that the instructions differed slightly for each participant.

would use the keywords variant of the instructions, see Figure 7-11. Note that 5 tasks always used the Java instructions, and these 5 tasks would either be grouped together at the beginning or at the end. The remaining 10 tasks used either pseudocode or keywords.

The tasks are listed on the next page. The highlighted expressions in each task represent the text which was *not* shown to the user. This text was replaced with a text field prompting the user to supply an expression of their own (see Figure 7-11).


Scenario 1 / 15

```
public boolean isVowel(char c) {
    String vowels = "aeiou";
    return -1 != [                              ] Submit ;
}
```

- Infer what the program does at the given point...
  - Write ***keywords*** that suggest the proper code
  - Don't look online (guess parts you don't know)
  - Do the best you can, nobody will know this is you

Figure 7-11: Sample task using the *keywords* variant of the instructions.

1	// convert "1 2 3 4" --> "1,2,3,4"
	public String spacesToCommas(String message) {
	  String space = " ";
	  String comma = ",";
	  return **message.replaceAll(space, comma)**;
	}
		2	// convert "100" --> 100
			String input = "100";
			int output = **new Integer(input)**;

3	// make sure that the list has no more than the
	// given number of elements
	public void trimSize(List list, int noMoreThan) {
	  while (list.size() > noMoreThan) {
	    list.remove(list.length() - 1);
	  }
	}
		4	public boolean isFruit(String food) {
			  Set<String> fruits =
			        new HashSet<String>(getFruitList());
			  return **fruits.contains(food)**;
			}

5	public boolean isVowel(char c) {
	  String vowels = "aeiou";
	  return -1 != **vowels.indexOf(c)**;
	}
		6	Map<Integer, String> numberNames =
			        new HashMap<Integer, String>();
			Integer key = 3;
			String value = "Three";
			// make numberNames have the entry:
			// 3 --> "Three"
			numberNames.put(key, value);

7	public int absoluteValue(int x) {
	  // use standard java method to
	  // return the absolute value of x
	  return **Math.abs(x)**;
	}
		8	public Vector<String> getTokens(String message) {
			  Vector<String> tokens = new Vector<String>();
			  StringTokenizer st =
			        new StringTokenizer(message, " ");
			  while (st.hasMoreTokens()) {
			    tokens.add(st.nextToken());
			  }
			  return tokens
			}

9	// count the a's
	String message =
	        "how many a's are in this message?";
	int count = 0;
	for (int i = 0; i < message.length(); i++) {
	  char c = **message.charAt(i)**;
	  if (c == 'a') {
	    count++;
	  }
	}
		10	// example output:
			// autoexec.bat
			// config.sys
			// Documents and Settings
			// Windows
			public void ls(File dir) {
			  for (File f : dir.listFiles()) {
			    System.out.println(f.getName());
			  }
			}

11	public String repeatString(String s, int
	thisManyTimes) {
	  StringBuffer buf = new StringBuffer();
	  for (int i = 0; i < thisManyTimes; i++) {
	    buf.append(s);
	  }
	  return buf.toString();
	}
		12	public List<String> getLines(BufferedReader in) {
			  List<String> lines = new Vector<String>();
			  while (in.ready()) {
			    lines.add(in.readLine());
			  }
			  return lines;
			}

13	public void logMessage(String message) {
	  PrintWriter log = new PrintWriter(
	        new FileWriter("log.txt", true));
	  log.println(message);
	  log.close();
	}
		14	// convert "HeLLo WoRlD" --> "hello world"
			String input = "HeLLo WoRlD";
			String output = **input.toLowerCase()**;

15	String filename = "input.txt";
	BufferedReader in = **new BufferedReader(new FileReader(filename))**;
	in.read(...);
	in.close();		

Evaluation:

Each user's response to each task was recorded, along with the instructions shown to the user for that task. Recall that users who omitted any responses, or supplied garbage answers, were removed from the data.

Tasks 1 and 3 were also removed from the data. Task 1 was removed because it is inherently ambiguous to the Keyword Tree and Bottom-up algorithms, since they do not take word order into account. Task 3 was removed because it requires a literal, and neither algorithm currently handles literals.

The remaining responses were provided as keyword queries to the Eclipse Plugin in the context of each task. The resulting suggestions were recorded from both the Keyword Tree and Bottom-up algorithms. Note that all the tasks were included in a single file as separate functions. The size of the resulting model was at least $|F| = 2281$ and $|T| = 343$ for each task, plus a few functions to model the local variables in each task.

7.3.3 Results

When asked to write Java code, users wrote syntactically and semantically correct code 53% of the time. This number is used as a baseline benchmark for interpreting the results of the algorithms. Note that `log.write(message)` was considered semantically correct for task 13, since the task did not make it obvious that a newline character was required.

The Keyword Tree algorithm translated 50% of the responses to semantically correct Java code. Note that this statistic includes *all* the responses, even the responses when the user was asked to write Java code. This makes sense because the user could enter syntactically invalid Java code, which may be corrected by the algorithm. In fact, the Keyword Tree algorithm improved the accuracy of Java responses from the baseline 53% to 67%. The accuracies for translating pseudocode and keywords were lower, at 42% and 41% respectively.

The Bottom-up algorithm did better, translating 59% of responses to semantically

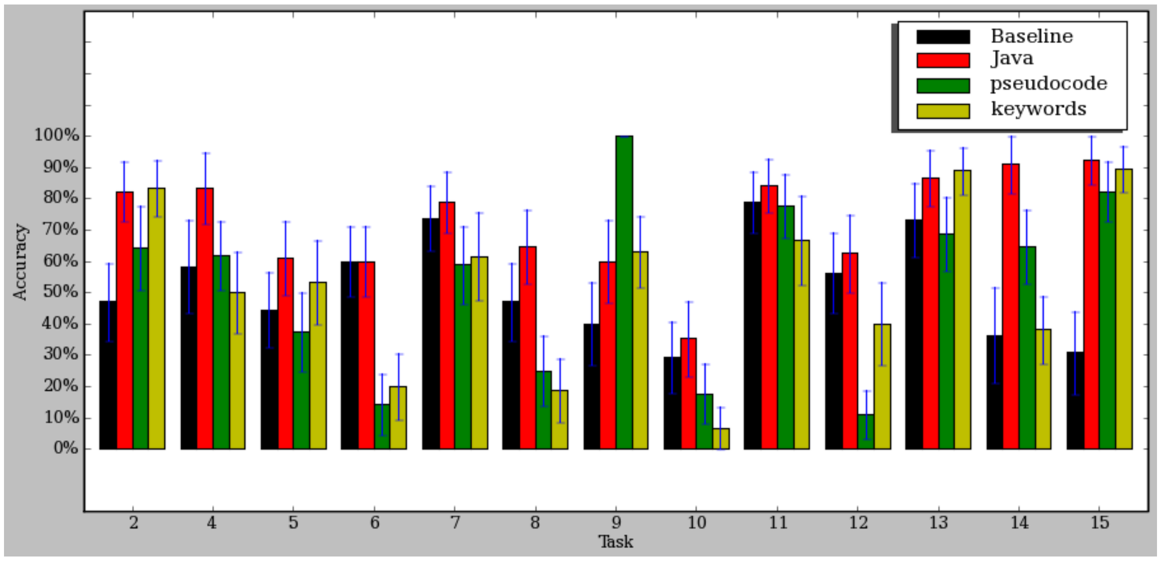


Figure 7-12: Accuracy of Bottom-up algorithm for each task, and for each instruction type, along with standard error. The "Baseline" refers to Java responses treated as Java, without running them through the algorithm.

correct Java code. The Java responses were improved from 53% to 71%, and the pseudocode and keywords were both 53%. This is encouraging, since it suggests that users of this algorithm can obtain the correct Java code by writing pseudocode or keywords as accurately as they can write the correct Java code themselves.

The Bottom-up algorithm performed much better than the Keyword Tree algorithm. One reason is that the Bottom-up algorithm can infer unnamed functions. For instance, one user typed **message[i]** for task 9, which the Bottom-up algorithm correctly translated to `message.charAt(i)` by inferring the `charAt` function. The Keyword Tree algorithm could not generate any code for this response.

The remaining results and analysis are limited to the Bottom-up algorithm. A breakdown of the accuracies for each task, and for each instruction type, are shown in Figure 7-12.

The following table shows a random sample of responses for each instruction variant that were translated correctly and incorrectly. Subscripts indicate the task associated with each sample. The responses were quite varied, though it should be noted that many responses for pseudocode and keywords were written with Java style syntax.

instructions	translated correctly	translated incorrectly
Java	Math.abs(x)$_7$	return(x>=0?x;-x);$_7$
	input.toInt()$_2$	tokens.append(st.nextToken())$_8$
	tokens.add(st.nextToken())$_8$	buf.add(s)$_{11}$
pseudocode	letter at message[i]$_9$	(x < 0) ? -x : x$_7$
	System.out.println(f.name())$_{10}$	lines.append (in.getNext());$_{12}$
	input.parseInteger()$_2$	input.lowercase();$_{14}$
keywords	vowels search c$_5$	Add s to buf$_{11}$
	lines.add(in.readLine())$_{12}$	in readline insert to lines$_{12}$
	buf.append(s);$_{11}$	print name of f$_{10}$

7.3.4 Discussion

It is useful to look at some of the incorrectly translated responses in order to get an idea for where the algorithm fails, and how it could be improved.

A Priori Word Weights

The Bottom-up algorithm translated **print name of f** in task 10, to `Integer.valueOf(f.getName())`. Apparently it could not come up with an expression that explained all the keywords, so it settled for explaining **name**, **of**, and **f**, and leaving **print** unexplained. However, **print** is clearly more important to explain than **of**.

One possible solution to this problem is to give a priori weight to each word in an expression. This weight could be inversely proportional to the frequency of each word in a corpus. It may also be sufficient to give stop words like **of**, **the**, and **to** less weight.

A Priori Function Weights

The response **println f name** in task 10 was translated to `System.err.println(f.getName())`. A better translation for this task would be `System.out.println(f.getName())`, but the algorithm currently has no reason to choose `System.out` over `System.err`. One way to fix this, along a similar vein to word weight, would be to have a priori function weights. These weights could also be derived from usage frequencies over a corpus, assuming `System.out` is used more more frequently than `System.err`.

Of course, these function weights would need to be carefully balanced against the cost of inferring a function. For instance, the input **print f.name** in task 10 was translated to `new PrintWriter(f.getName())`, which explains all the keywords, and doesn't need to infer any functions. In order for the algorithm to choose `System.out.print(f.getName())`, the cost of inferring `System.out`, plus the weight of `print` as an explanation for the keyword **print** would need to exceed the weight of `new PrintWriter` as an explanation for **print**.

Spelling Correction

Many users included **lowercase** in their response to task 14. Unfortunately, the algorithm does not see a token break between **lower** and **case**, and so it does not match these tokens with the same words in the desired function `toLowerCase`. One solution to this problem may be to provide spelling correction as discussed in the Web Prototype User Study. That is, a spelling corrector would contain `toLowerCase` as a word in its dictionary, and hopefully **lowercase** would be corrected to `toLowerCase`. One problem is the missing letters **t** and **o** at the beginning, which may confuse some spelling correction algorithms if they give more weight to characters at the beginning. As future work, it may useful to collect real word samples of programming misspellings to inform the design of a spelling corrector for this purpose.

Synonyms

Another frequent problem involved users typing synonyms for function names, rather than the actual function names. For instance, many users entered **append** instead of **add** for task 8, e.g., **tokens.append(st.nextToken())**.

An obvious thing to try would be adding `append` to the label of the function `add`, or more generally, adding a list of synonyms to the label of each function. To get a feel for how well this would work, synonyms were added to each function using WordNet [12] as a thesaurus, and reran the experiments. This improved some of the translations, but at the same time introduced ambiguities in other translations. Overall, the accuracy decreased slightly from 59% to 58%.

An example of where the synonyms helped was in translating inputs like **tokens.append(st.nextToken)** for task 8, into `tokens.add(st.nextToken())`. An example of where the synonyms caused problems was task 6, where the input **numberNames.put(key,value)** was translated erroneously as `String.valueOf(numberNames.keySet())`. The reason for this translation is that `put` is a synonym of `set` in WordNet, and so the single function `keySet` explains both the keywords **put** and **key**. Also note that the algorithm uses a heuristic which prefers fewer function calls, and since `String.valueOf(numberNames.keySet())` has only three function calls (where `String.valueOf` is a single static function), this interpretation is favored over `numberNames.put(key, value)` which has four function calls.

Another problem was caused in task 12, where inputs like **lines.add(in.nextLine())** were translated into `lines.add(in)`, instead of `lines.add(in.readLine())`. This happened because `line` was added as a synonym for `lines`, allowing the single function `lines` to explain two keywords from the input.

The latter problem with `lines` and `line` seems like it could be solved by using a stemmer, but in general, it is difficult to anticipate all of the repercussions of changing the heuristics in the algorithm. Developing a principled manner in which to adjust these heuristics is an important area for future work, and is discussed more in the next chapter.

Chapter 8

Discussion

Overall, the algorithms seem to work well, and the interface seems intuitive to use. The Reference algorithm achieves high accuracy in the Web domain, even without providing instructions about how to form keyword queries. Similar accuracies may be expected in other small domains.

The Keyword Tree and Bottom-up algorithms achieve accuracies comparable with writing unassisted Java code, even with extremely loose instructions about how to use keyword queries. Presumably better instructions would help, like telling users that the system does not currently support literals or control flow. Users may also perform better when using the actual plugin, since they would be presented with feedback, whereas users in the study were given no feedback about the quality of their responses.

However, these evaluations have also exposed a number of limitations in the algorithms. It remains to be seen which limitations may be overcome with changes to the algorithms, and which limitations are inherent to keyword programming. Some of these issues are discussed below.

8.1 Consistency

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

Some context changes are visible to the user, like a webpage containing a “refresh” button, but some changes are invisible, like an API adding or changing functions. Note that API also change in formal languages, although the changes may be more apparent in those cases since they are likely to result in syntax errors. On the other hand, syntax errors are probably not as good as inputs continuing to work, even with the changes to the API, which is a possibility with keyword queries. Whether context changes are more or less of a problem with keyword programming remains to be seen, and is probably domain specific.

8.2 Ambiguity

Another potential issue with keyword programming is ambiguity. The whole point of formal programming languages is to eliminate ambiguity, so reintroducing ambiguity to programming may cause problems.

Ambiguities may not be a problem if they only arise when keyword queries are less verbose than formal language commands, since the user always has the option of being more verbose. However, some ambiguities may be impossible to resolve, i.e., keyword queries may not be expressive enough to disambiguate certain expressions. The open question here is whether such ambiguities are common, and they are best resolved. One option to explore is presenting the user with a dropdown containing possible interpretations, and then learning the users preferences so that the same commands do not require disambiguation in the future.

Another issue related to ambiguity is the phrase structure requirement in the Reference and Keyword Tree algorithm. This requirement has advantages and disadvantages. The primary advantage is that in principle, it allows users to resolve more ambiguities. However, it is less clear how often this is useful in practice, and whether this is the best approach to resolve these ambiguities. The disadvantage of phrase structure is that it imposes an additional constraint on user input, and perhaps more importantly, it makes the algorithmic problem more difficult. The algorithms which pay attention to phrase structure are the slowest.

8.3 Heuristics

So far the discussion has focused mainly on problems faced by the user, but keyword programming also poses problems for system designers who wish to incorporate keyword programming into their interfaces. In particular, the translation algorithms use many weights and heuristics, which may be difficult to tune to each domain.

The general strategy for weights used in this thesis has been an additive point system: each keyword is generally worth 1 point, and various other features either add or subtract points.

Some feature weights are orders of magnitude smaller than others, in order to mitigate conflicts. For instance, the Eclipse Plugin favors functions with small names by subtracting a small number of points proportional to a function name's length, when it is included in a function tree. The idea is that functions with small names have a higher a priori probability of being called. However, explaining an addition keyword is always better than using a small function.

However, other weights are in the same order of magnitude, and these do interact. For instance, the Eclipse Plugin favors small function trees by including a cost to calling each function. However, it also favors functions for which all the keywords in the label are present in the input, and this cost is the same order of magnitude. These weights are difficult to tune, since different weights fix some translations, but break others.

Still other weights have not been added to the system, but seem like they would be important. For instance, the idea of giving different weights to different keywords in the input, so that some keywords are worth more to explain than others. Also, the idea of giving a priori weights to functions based on their probabilities of use, rather than simply on the length of their names. A third idea is to add weight when keywords in the input are in the same order as keywords in a function.

A compounding factor is the issue of spelling correction, and synonyms. Spelling correction becomes an issue of weights when string literals are allowed to be unquoted, since a misspelled word may be more likely to be a string literal than a correctly spelled word. Synonyms are also an issue, since the explanatory power of a function should probably be less if it uses a synonym rather than the actual function name.

All of these ideas add complexity to the system of weights, and the primary focus of future work in keyword programming will be to derive a principled way of assigning weights. An interesting open question along these lines is what impact changes to an API will have on the optimal weights, and whether the weights can be derived independently from the API and programming domain.

Chapter 9

Conclusions & Future Work

This thesis explores three different paths for using keyword programming: application scripting, business process scripting, and Java programming.

9.1 Application Scripting

The results from the web prototype user study suggest that the system is usable for applications, provided that end users are familiar with the terminology of the application (as they were with the web). In the web domain at least, it seems that end users can use the system without any instructions or training. However, the studies also suggest that the reference algorithm does not scale to larger domains. This inspired work on the later two algorithms targeted at the Java domain.

As future work, I would like to add new features to the algorithms. In particular, I would like to support variable declarations which are referenced in the same command. This may allow for for-loop constructs.

Some of the other goals after the web prototype study are already underway, including integrating this system into Chickenfoot; I would like to see if end-users without JavaScript experience can create simple scripts. In Word, I would like to test how keyword commands compare with navigating a large menu system. With the new algorithms, this should be a feasible prototype to create and test.

I would also like to explore the potential of using the translation system as a generic back end for a speech recognition system. I believe that the general framework may make it easy to expose core sets of functionality of various applications to end-users, in particular, entertainment systems (either at home or in the car).

9.2 Business Process Scripting

Although I did not conduct a study of the Koala system, the system was release to the "IBM early adopter community" in December of 2006, and has garnered 600+ users and 300+ scripts by the time of this writing (which is May 2007). As future work, I would like to explore the impact of this system on end user work flow when it comes to business processes. I have already benefited from the system myself; when I agreed to join the internship program again next summer, I received an e-mail with web-based instructions informing me on how to submit my resume. These instructions were a Koala script.

9.3 Java Programming

This thesis presents a novel technique for code completion in Java, in which the user provides a keyword query and the system generates type-correct code that matches those keywords. I also provide two algorithms which perform this search efficiently. Using example queries automatically generated from a corpus of open-source software, I found that the type constraints of Java ensure that a small number of keywords is often sufficient to generate the correct method calls.

Although these algorithms are faster, I have not implemented all the whistles and bells of the reference algorithm, including spell correction and stemming. I would also like to add more mappings to Java constructs— some of my ideas in this regard are discussed in the *Model* chapter. The largest piece of future work in this domain will be incorporating a priori weights for words and functions, and figuring out a principled manner for selecting these weights.

I also plan to apply keyword programming to other programming languages, so that multiple-language programming becomes less arduous. Ideally, a keyword query like **add `x` list** should generate appropriate code to add `x` to `list`, regardless of the particular language or API in use.

The long-term goal for this work is to simplify the usability barriers of programming, such as forming the correct syntax and naming code elements precisely. Reducing these barriers will allow novice programmers to learn more easily, experts to transition between different languages and different APIs more adroitly, and all programmers to write code more productively.

Bibliography

[1] *Your wish is my command: programming by example.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[2] Bruce W. Ballard and Alan W. Biermann. Programming in natural language: "NLC" as a prototype. In *ACM 79: Proceedings of the 1979 annual conference*, pages 228–237, New York, NY, USA, 1979. ACM Press.

[3] Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. DocWizards: a system for authoring follow-me documentation wizards. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 191–200, New York, NY, USA, 2005. ACM Press.

[4] Blacktree. Quicksilver. http://quicksilver.blacktree.com/.

[5] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 163–172, New York, NY, USA, 2005. ACM Press.

[6] Amy Bruckman. Community support for constructionist learning. *Comput. Supported Coop. Work*, 7(1-2):47–86, 1998.

[7] Amy Bruckman and Elizabeth Edwards. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 207–214, New York, NY, USA, 1999. ACM Press.

[8] Apple Computer. Automator. http://www.apple.com/macosx/features/automator/.

[9] Matthew Conway, Randy Pausch, Rich Gossweiler, and Tommy Burnette. Alice: A rapid prototyping system for building virtual environments. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 2, pages 295–296, April 1994.

[10] Allen Cypher. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[11] Martin Erwig, Robin Abraham, Irene Cooperstein, and Steve Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 136–145, 2005.

[12] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[13] Google. Calendar. http://www.google.com/calendar/.

[14] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press.

[15] C. Kelleher, D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, and R. Pausch. Alice2: Programming without syntax errors. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, New York, NY, USA, 2002. ACM Press.

[16] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, 2005.

[17] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Hugo Liu and Henry Lieberman. Metafor: visualizing stories as code. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 305–307, New York, NY, USA, 2005. ACM Press.

[19] Hugo Liu and Henry Lieberman. Programmatic semantics for natural language interfaces. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1597–1600, New York, NY, USA, 2005. ACM Press.

[20] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press.

[21] L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215, 1981.

[22] P. Miller, J. Pane, G. Meter, and S. Vorthmann. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.

[23] Robert C. Miller. *Lightweight Structure in Text.* PhD thesis, Carnegie Mellon University, 2002.

[24] John F. Pane, Brad A. Myers, and Chotirat Ann Ratanamahatana. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comput. Stud.*, 54(2):237–264, 2001.

[25] Alok Parlikar, Nishant Shrivastava, Varun Khullar, and Sudip Sanyal. NQML: Natural Query Markup Language. In *Natural Language Processing and Knowl-*

edge Engineering, 2005. IEEE NLP-KE '05. Proceedings of 2005 IEEE International Conference on, pages 184–188. IEEE Computer Society, 2005.

[26] Martin Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[27] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. NaturalJava: a natural language interface for programming in Java. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 207–211, New York, NY, USA, 2000. ACM Press.

[28] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA, 2006. ACM Press.

[29] Jean E. Sammet. The use of English as a programming language. *Commun. ACM*, 9(3):228–230, 1966.

[30] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[31] Allen M. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.