

# Keyword Command Completion in Java

Greg Little

MIT CSAIL  
glittle@gmail.com

Robert C. Miller

MIT CSAIL  
rcm@mit.edu

## Abstract

\* Abstract - point out key contributions - show that it's possible to type a query, and search over the space of possible code (show the space is small? not small, just searchable) - we show that the information content of java code is relatively small - punctuation is not super important - we present two approaches for searching this space - we discuss heuristics to improve this search - we evaluate our approach on an artificial as well as people-generated corpus

**Categories and Subject Descriptors** CR-number [*subcategory*]: third-level

**General Terms** Experimentation, Languages

**Keywords** Java, Auto-complete

## 1. Introduction

We are working toward advancing code completion technology in Java. Imagine a programmer is adding lines of text from a stream called “in” to a list called “lines”. We want to let them to enter:

**add line**

and have the computer suggest:

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

More generally, we would like to take a string of keywords (a Keyword Command) and treat them as a search query over the space of valid expressions, given the current context in the code. Note that order is unimportant in this query, and some keywords may be ommitted.

Our current prototype is an Eclipse Plugin that allows the user to enter a Keyword Command directly into their source code, press a hot-key, and have the keywords replaced with compilable code that is a good match for the keywords.

We believe this system decreases cognitive load in several ways. First, Keyword Commands are shorter than code, and easier to type. Second, the user does not need to recall all the lexical components involved in an expression, since the computer may be able to infer some of them. Third, the user does not need to type (or even know) the syntax for calling methods.

In this paper, we propose two algorithms for finding Keyword Command completions quickly. This work builds on Keyword Commands [?], which is a system for translating arbitrary input sequences into function calls over some API. But the algorithm requires the API to be very small, on the order of 20 functions. The primary contribution of this paper is the application of this technique to Java, with two algorithms that perform efficiently on API's of over 2000 functions.

This work also builds on the Koala “Slop Interpreter” [?]. The key idea here is to enumerate all the possible commands, and then match them to the entire input sequence (as opposed to trying to build a tree out of the input sequence). Of course, Koala works in the web domain, and the number of possible actions on a webpage is relatively small. A general purpose programming language like Java has many more possibilities, making the algorithmic problem more difficult.

This work is similar to Prospector [?], which takes two Java types as input, and returns snippets of code that convert from one type to the other. However, this system mines snippets from a large corpus of existing code, and is most useful when the creation of a particular type from another type is non-obvious (i.e. you can't simply pass it to the constructor, and other initialization steps may be involved).

XSnippet retrieves snippets based on context, e.g., all the available types from local variables. However, the query is still for a snippet of code that achieves a given type, and the intent is still for large systems where the creation of certain types is nontrivial.

The key differences in our approach are:

1. The input is not restricted to a type, although it *is* constrained by types available in local context. Also, the output code may be arbitrary, not just code to obtain an object of a certain type. For instance, you could use our system to enter code on a blank line, where there is no restriction on the return type.

2. Our system uses a guided search, based on the keywords you provide. These keywords can match methods, variables and fields that may be used in the expression.
3. The system *generates* code, and does not require a corpus of existing code to mine for snippets. In particular, users could benefit from our system in very small projects that they are just starting.

Our key contributions are:

- Two algorithms for translating Keyword Commands into Java code efficiently.
- An Eclipse Plugin that allows users to perform Keyword Command completions in their Java source code.
- An evaluation of the algorithms on an artificial corpus, as well as a user supplied corpus.

## 2. Model

The scenario we want to model is the following: the user is at some location in their source code, and they have entered a Keyword Command. They intend the keywords to produce a java expression. In order to find the expression, we need to model the available methods, fields and local variables, and how they fit together using Java's type system.

Our model consists of types, labels, and functions.

### 2.1 Type Set: $T$

We have a set of types  $T$ . Each type is represented by a unique name. For instance, Java's integer type is represented as `int`.

We include all the primitive types and classes referenced in the current source file, as well as object types that are accessible with at most 2 method calls. This number is arbitrary, but all the experiments on our algorithms assume that we are building nested expressions with depth at most 2, e.g. `func0(func1(func2()))`.

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  $\top \in T$ , and define  $sub(\top) = T$ .

### 2.2 Label Set: $L$

We keep a set of labels  $L$ . Each label is a sequence of keywords. We use labels to represent method names, so that we can match them against the keywords in a Keyword Command.

The get the keywords from a method name, we break up the camel-case words. For instance, the method name `currentTimeMillis` is represented with the label (**current**, **time**, **millis**). Note that capitalization is unimportant for keywords.

### 2.3 Function Set: $F$

Let  $F$  denote a set of functions. Functions are used to model anything we want to match against the user's Keyword Command, including methods, fields, and local variables.

We define a function as an  $n$ -tuple in  $T \times L \times T \times \dots \times T$ . The first  $T$  is the return type, followed by the method name's label, and all the parameter types. For 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.

### 2.4 Function Tree

The purpose of defining types, labels and functions is to model java expressions that the user may intend with their Keyword Command. We model a java 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 an  $n$ -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. We'll require that the number of children in a node be equal to the number of parameter types of the function, i.e.,  $|children(n)| = |params(func(n))|$ . We'll also require that the return types from the children "fit" into the parameters, i.e.,

$$\forall_i ret(func(children(n)_i)) \in sub(params(func(n))_i).$$

Note that in the end, the system will render the Function Tree as a syntactically correct Java expression.

### 2.5 Java Mapping

We now provide the particulars for mapping various Java elements to  $T$ ,  $L$  and  $F$ . Most of these mapping are natural and straightforward.

#### 2.5.1 Primitive Types:

Primitive types are modeled as types in  $T$ , using their traditional names (`int`, `long`, `double`, etc...). Note that because of boxing and unboxing in Java 1.5, we include `java.lang.Integer` in  $sub(int)$ .

#### 2.5.2 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  $c$ .

#### 2.5.3 Methods:

Methods are modeled as functions that take their 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)
```

#### 2.5.4 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)
```

#### 2.5.5 Local Variables:

Local variables are simply 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))`.

#### 2.5.6 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)
```

#### 2.5.7 Members:

Member methods and fields of the class containing the Keyword Command 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))`.

#### 2.5.8 Statics:

The Java syntax for calling methods does not usually require writing type names, which is why we do not include type names in the label when modeling methods. However, type names are used to disambiguate static methods and fields, since they have no receiver object.

To support this syntax, we model static methods and fields with an additional function. Our strategy is to omit the object as the first parameter, but add the class name as part of the function label. 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)
```

An alternate approach is to have two types, instead of two functions. For instance, we could add the additional type

`static: java.lang.Math to  $T$` , and then have one function for `sin`, namely

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

We would then make `java.lang.Math` a subtype of `static: java.lang.Math`, and we would add a special constructor function for static types, like

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

#### 2.5.9 Generics:

We support generics explicitly, i.e., we create 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 we add both of these types to  $T$ . We also add 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 at a time.

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.

### 3. Problem

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

The input to the algorithm consists of  $T$ ,  $L$  and  $F$ , and some keywords (i.e. a Keyword Command). We also supply a desired return type, which we make as specific as possible given the source code around the Keyword Command. (If any type is possible, we supply  $\top$  as the desired return type).

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

Choosing a good similarity metric between a Function Tree and a Keyword Command is the real challenge. We need a metric that matches human intuition, as well as a metric that is easy to gauge algorithmically.

As a starting point, we use the simple metric where 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 metric varies somewhat between the two algorithms, and depends on their implementations and heuristics.

## 4. Algorithms

We have developed two different algorithms for solving this problem. The first algorithm uses a genetic algorithm to explore possible ways to arrange keywords into trees. The second algorithm is a bottom up algorithm that matches keywords against types they are likely to obtain.

The first algorithm is able to benefit from phrase structure information, but does not allow function inference. The second algorithm ignores the order of the keywords completely, and it can infer functions.

### 4.1 Algorithm 1: Keyword Tree

In the Keyword Tree algorithm, we define a genome that arranges keywords into a parse tree. We then score the parse tree based on the Function Trees we can create from it. Finally, we use a genetic algorithm to search over the space of possible parse trees for the one that can produce the best Function Tree.

#### 4.1.1 Genome

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

Consider that we have  $n$  keywords,  $k_1, k_2, \dots, k_n$ . To start, we assume that each keyword is a singleton tree. The genome consists of  $n - 1$  components  $c_1, c_2, \dots, c_{n-1}$ . Each component can be thought of as existing between two keywords. In particular, we 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  $right(c)$ .
- $order(c)$  is a real number between 0 and 1 that determines the order in which to apply the rules. We generate these randomly, and assume each  $order$  value will be unique.
- $index(c)$  is also number in the range 1 to  $n - 1$  that specifies where to insert nodes when we make one node the child of another node. If a node has  $n$  children, we insert at position  $index \bmod (n + 1)$ .

Here 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 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).

```

C ← {c1, c2, ..., cn-1}
for each c ∈ C
  if rule(c) = ↔
    do {
      then {combine(root(left(c)), root(right(c)))
      C ← C - {c}
    }
while C ≠ ∅
  comment: get component with smallest order
  c ∈ {a | a ∈ C and ∀i order(a) ≤ order(ai)}
  comment: apply the rule for c
  if rule(c) = ↗
    do {
      then {parent ← root(right(c))
      child ← root(left(c))
      else if rule(c) = ↘
        then {parent ← root(left(c))
        child ← root(right(c))
      i ← index(c) mod (|children(parent)| + 1)
      insert(parent, child, i)
      C ← C - {c}

```

Here is an example to illustrate. Consider the the following Keyword Command, 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:  $(\nearrow, 3, 2)$ ,  $(\leftrightarrow, 2, 3)$ , and  $(\nwarrow, 1, 1)$ . We can visualize these sitting between the keywords:

**boxes**  $(\nearrow, 3, 2)$  **add**  $(\leftrightarrow, 2, 3)$  **box**  $(\nwarrow, 1, 1)$  **b**

In the first pass, we use the components with a  $rule$  of  $\leftrightarrow$  to combine neighboring keywords into single nodes. We see such a component between **add** and **box**. We'll denote combining them into a single node by writing them together using camel-case, giving us:

**boxes**  $(\nearrow, 3, 2)$  **addBox**  $(\nwarrow, 1, 1)$  **b**

Next, we find the remaining component with the lowest  $order$ , which is  $(\nwarrow, 1, 1)$  between **addBox** and **b**. The  $rule$  is  $\nwarrow$ , so we make **b** a child of **addBox**. In this case there is only one place to insert it, so we can ignore the  $index$ . We'll denote making **a** the child of **b** with **b(a)**, giving us:

**boxes**  $(\nearrow, 3, 2)$  **addBox(b)**

The final component tells us to make **boxes** a child of **addBox**. Since **addBox** already has a child **b**, we use the  $index$  of 2 to determine where to insert the new child. We take  $index$  modulus one more than the number of children already present, which is written as  $2 \bmod (1 + 1)$ . This evaluates to 0, so we insert **boxes** as the new first child of **addBox**, giving us:

**addBox(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 shows us that the genome is expressive enough to build this tree from the keywords. We would like to know 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 we 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; we just set the *rule* for each of these components to  $\leftrightarrow$ ). Now we take any one of the edges connected to the root, and we find the corresponding component  $c$ . Then we set  $order(c)$  to the largest unused *order* value, and we set the component  $index(c)$  to the 0-based index of the child connected to this edge. Then we set *rule* to  $\nearrow$  if  $root(right(c))$  is the root node, and we set *rule* to  $\nwarrow$  if  $root(left(c))$  is the root node.

Next, we remove this edge from the tree and repeat the process (now we are allowed to 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 we tore it down. ■

#### 4.1.2 Fitness Function

A parse tree may correspond to several different Function Trees. Consider the parse tree from our previous example:

**addBox(boxes, b)**

This corresponds to the Function Trees:

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

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

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

We would like to say that the correspondence is better for **A** than for **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 we would like to say that **C** is worse because it includes an unmatched keyword **synchronized**.

We formalize this notion in the way we score a function  $f$  with a node  $n$ . First we 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 not shared. Note that  $key(n)$  and  $label(f)$  are lists, and might contain repeated keywords. If a keyword  $k$  appears  $x$  times in  $key(n)$  and  $y$  times in  $label(f)$ , then we award  $\min(x, y)$  points for this keyword, and subtract  $0.01 * (\max(x, y) - \min(x, y))$  points.

To determine the score for the entire parse tree, we define  $score(n)$  as the score for the parse tree rooted at  $n$ . We also define  $score(n, t)$  as the score of the best Function 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 we are not always

able to find a valid Function Tree that corresponds with the parse tree.

We calculate  $score(n)$  and  $score(n, t)$  recursively. To process node  $n$ , we first process the children of  $n$ . Next, for each function  $f$ , we will calculate a score  $s$ . We will update  $score(n)$  and  $score(n, ret(f))$  based on  $s$ .

We initialize  $s$  with the score calculated between  $label(f)$  and  $key(n)$ , as described above. To speed up the algorithm, we skip the function if  $s \leq 0$ . Next, we add a point to  $s$  for each genome component with rule  $\leftrightarrow$  that went into forming  $n$ .

We then add 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 we add  $\max_{t \in sub(p)} score(n', t)$  to  $s$ , in addition to a point for the genome component that attached this child.

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

Finally, we update  $score(n)$  to  $\max(s, score(n))$ . We also update  $score(n, ret(f))$  to  $\max(s, score(n, ret(f)))$ , but only if we found children returning the proper types for each parameter of  $f$ .

#### 4.1.3 Genetic Algorithm

We create an initial population of 100 genomes, but each subsequent generation has only 10 genomes. We run the algorithm 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 randomly selected parents: the previous generation is put in order, and parents are chosen using a random index from  $\lfloor N(0, 3^2) \rfloor \bmod 10$ .

New genomes are created using cross-over and mutation. We choose one cross-over point. 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.

#### 4.1.4 Extraction

After running the genetic algorithm, we end up with a good parse tree. Now we need to extract the best Function Tree from it. We do this 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, we use a simple recursive algorithm 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 we repeat this process recursively to find the

best function that achieves the desired parameter type from each child.

#### 4.1.5 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 "hello world"**, and expect the system to generate `System.out.println("hello world")`, since the system cannot infer the field `System.out`. Our next algorithm overcomes both of these limitations.

## 4.2 Algorithm 2: Bottom-Up

This algorithm can be thought of as a dynamic program where we are filling out a table of the form  $func(t, i)$ , which tells us which function to use to achieve type  $t$ , if the Function Tree can be at most height  $i$ . It also tells us the score we expect to achieve with the resulting Function Tree.

Calculating  $func(t, 0)$  for all  $t \in T$  is relatively easy. We only consider functions that take no parameters, since our tree height is bounded by 0. Then for each such function  $f$ , we give it some score, and we associate this score with  $ret(f)$ . Then for each  $t \in T$ , we update  $func(t, 0)$  with the best score associated with any subtype of  $t$ .

Instead of a scalar value for the score, we use an Explanation Vector. We'll explain what this is before talking about the next iteration of the dynamic program.

### 4.2.1 Explanation Vector

The idea of the Explanation Vector is to encode how well we have explained the input keywords. If we have  $n$  keywords  $k_1, k_2, \dots, k_n$ , then the Explanation Vector has  $n+1$  elements  $e_0, e_1, e_2, \dots, e_n$ . Each element  $e_i$  represents how well we have explained the keyword  $k_i$  on a scale of 0 to 1; except  $e_0$ , which represents explanatory power not associated with any particular keyword. When we add two Explanation Vectors together, we ensure that the resulting elements  $e_1, e_2, \dots, e_n$  are capped at 1, since the most we can explain a particular keyword is 1.

Before we do anything else, we calculate an Explanation Vector  $expl(f)$  for each function  $f \in F$ . In the common case, We set  $e_i$  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:

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

We set  $e_0$  to  $-0.01x$ , where  $x$  is the number of words appearing in either the input or  $label(f)$ , but not both. In this case, we set  $e_0$  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, we distribute our explanation of the word **node** between them:

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

In general, we set  $e_i = \max(\frac{x}{y}, 1)$ , 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 we set  $e_0$  to  $-0.03$ , since there are three words that appear in the input, but not in the function label (we include one of the **node** keywords in this count, since it only appears once in the label).

### 4.2.2 Next Iteration

In the next iteration, we are trying 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  $func(p, i-1)$ .

We can do this, but there is a problem. We no longer know that we have the optimal Explanation Vector possible for this function at this height. Consider the following input:

**add x y**

and assume we have the functions:

(int, (add), int, int),

(int, (x)), and

(int, (y)),

If we look in  $func(int, 0)$ , we are likely to see either (int, (x)), or (int, (y)). Let's assume it is (int, (x)). Now consider what happens in the next iteration when we are processing the function (int, (add), int, int). We take the Explanation Vector  $(-0.02, 1_{add}, 0_x, 0_y)$ , and we add the Explanation Vector found in  $func(int, 0)$ , which is  $(-0.02, 0_{add}, 1_x, 0_y)$ . This gives us  $(-0.04, 1_{add}, 1_x, 0_y)$ .

Now we want to add the Explanation Vector for the second parameter, which is also type int. We look in  $func(int, 0)$  and find  $(-0.02, 0_{add}, 1_x, 0_y)$  again. When we add it, we get  $(-0.06, 1_{add}, 1_x, 0_y)$ , since the keyword components are capped at 1.

But what if we had found the Explanation Vector for  $(\text{int}, (y))$ ? Then we could have gotten  $(-0.06, 1_{\text{add}}, 1_x, 1_y)$ , which is better.

To get around this problem, we store the top  $x$  functions at each  $\text{func}(t, i)$ , where  $x$  is an arbitrary constant. In our case, we chose 3, except in the case of  $\text{func}(\dots\text{Object}, i)$ , where we use keep the top 5.

Now when we are considering function  $f$  at height  $i$ , and we are adding Explanation Vectors for the parameters, we are greedy: we add the Explanation Vector that increases our final vector the most, and then we move on to the next parameter. Note that if our parameter type is  $p$ , we consider all the Explanation Vectors in each  $\text{func}(p, j)$  where  $j < i$ .

### 4.2.3 Extraction

After we have run the dynamic program to some arbitrary height (in our case, 2), we need to extract a Function Tree.

We use a greedy recursive algorithm which takes the following parameters: a desired return type  $t$ , a maximum height  $h$ , and an Explanation Vector  $e$  (representing what we have explained so far). The function returns a new Function Tree, and an Explanation Vector. Note that we sort the parameters such that the most specific types appear first.

```

procedure EXTRACT TREE( $t, h, e$ )
  for each  $f \in \text{func}(t, i)$  where  $i \leq h$ 
    comment: create Function Tree node
     $n \leftarrow (f)$ 
     $e_n \leftarrow e + \text{expl}(f)$ 
    comment: put most specific types first
     $P \leftarrow \text{SORT}(\text{params}(f))$ 
    do for each  $p \in P$ 
      do  $\begin{cases} n_p, e_p \leftarrow \text{EXTRACT TREE}(p, i-1, e_n) \\ \text{comment: add } n_p \text{ as a child of } n \\ n \leftarrow \text{append}(n, n_p) \end{cases}$ 
      if  $e_n > \text{best}_e$ 
        then  $\begin{cases} \text{best}_e \leftarrow e_n \\ \text{best}_n \leftarrow n \end{cases}$ 
  return  $(\text{best}_n, \text{best}_e)$ 

```

## 5. Eclipse Plugin

With the Eclipse Plugin installed, the user may enter Keyword Commands directly into their source code. Here we show 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 a hot-key. We use Ctrl-I. When they press the hot-key, the system does three things:

1. It creates the models  $T$ ,  $L$ , and  $F$  based on the classes named in the source file, and the local variables in the current context. For instance, the system sees the class name `List<String>`, so it adds all the methods and fields associated with `List<String>`. It also see the local variable `lines`, and adds the function `(...List<String>, (lines))`.
2. It figures out where the Keyword Command 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 the Keyword Command. 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 use one of our 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 the source code with Java syntax as `lines.add(in.readLine())`:

```

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. In the future, we would like to support an auto-complete style dropdown with a list of results to choose from, or integrate the system with auto-complete itself.

## 6. Evaluation

We evaluate the the plugin and the algorithms on an artificial corpus, as well as a user supplied corpus.

### 6.1 Artificial Corpus

The artificial corpus consists of a variety of open source Java projects. We create artificial Keyword Commands by finding method calls and obfuscating them (removing punctuation and rearranging keywords). We then pass these keywords to the Plugin, and record whether it generates the original method call.

#### 6.1.1 Projects

We selected 14 projects from popular open source web sites, including sourceforge.net, codehaus.org, and objectweb.org. Projects were selected based on popularity, and our ability to compile them using Eclipse. Our projects include:

1. **Azureus**, an implementation of the BitTorrent protocol.

| Project    | Class Files | LOC    | Test Sites |
|------------|-------------|--------|------------|
| Azureus    | 2277        | 339628 | 75909      |
| Buddi      | 128         | 27503  | 7221       |
| CAROL      | 138         | 18343  | 2262       |
| Dnsjava    | 123         | 17485  | 2576       |
| Jakarta CC | 41          | 10082  | 1371       |
| jEdit      | 435         | 124667 | 24561      |
| jMemorize  | 95          | 14771  | 2196       |
| Jmol       | 281         | 88098  | 43216      |
| JRuby      | 427         | 72030  | 18458      |
| Radeox     | 179         | 10076  | 1115       |
| RSSOwl     | 201         | 71097  | 22393      |
| Sphinx     | 268         | 67338  | 12556      |
| TV-Browser | 760         | 119518 | 26481      |
| Zimbra     | 1373        | 256472 | 72916      |

**Table 1.** Project Statistics

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.
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 1 shows how many class files, and line-of-code each project contains. We also report the number of valid test sites, which we discuss in the next section.

### 6.1.2 Tests

Each test is conducted on a method call. We only consider method calls of depth 2 or less, and we make sure that they involve only the Java constructs outlined in our model. For example, these include local variables and static fields, but do not include literals or casts. Figure 1 shows a valid test site highlighted in the JRuby project.

To perform each test, we obfuscate the method call by removing punctuation, splitting camel-case identifiers, and rearranging keywords (while still maintaining phrase struc-

```

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 1.** Example Test Site

ture). We then treat this obfuscated code as a Keyword Command, which we pass on to the plugin. (Note that in these tests, we tell the plugin where the Keyword Command starts and ends).

For example, the method call highlighted in Figure 1 is obfuscated to the following Keyword Command:

**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 Command 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)`

We compare this string with the original source code (ignoring whitespace), and if it matches exactly, we record the test as a success. We also include other information about the test, including:

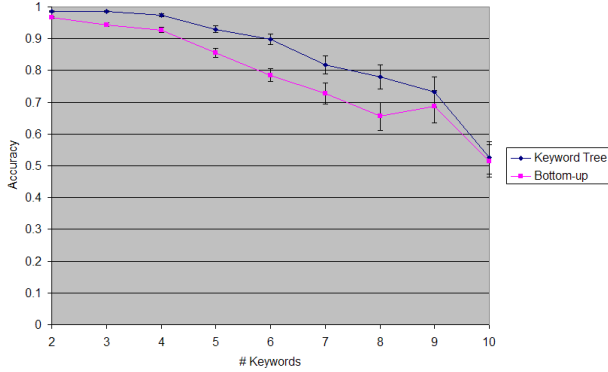
- **# Keywords:** the number of keywords in the Keyword Command.
- **time:** how many seconds the algorithm spent searching for a Function Tree.
- **Ambiguities:** the number of Function Trees constructable with the correct set of functions. If a function takes two parameters of the same type, then there are at least two ways to build the Function Tree.
- $|T|$ : 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.

### 6.1.3 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 we sampled 500 from each of them). This gives us 8000 test sites. For each test site, we tested both algorithms.

| # Keywords | Samples |
|------------|---------|
| 2          | 3667    |
| 3          | 1867    |
| 4          | 1030    |
| 5          | 569     |
| 6          | 373     |
| 7          | 187     |
| 8          | 122     |
| 9          | 86      |
| 10         | 37      |
| 11         | 30      |
| 12         | 19      |
| 13         | 7       |
| 14         | 1       |
| 15         | 4       |
| 17         | 1       |

**Table 2.** Samples given # Keywords



**Figure 2.** Accuracy given # Keywords

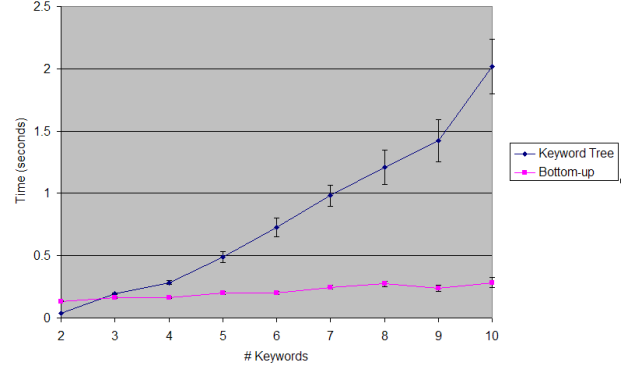
Table 2 shows how many samples we have for different Keyword Command lengths. Because we do not have many samples for large lengths, we will group all the samples of length 10 or more when we plot graphs against keyword length.

Figure 2 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 outperforms the Bottom-up algorithm. This may be due in part to the fact that the Keyword Tree algorithm takes advantage of phrase structure, which is something we preserve in the obfuscation.

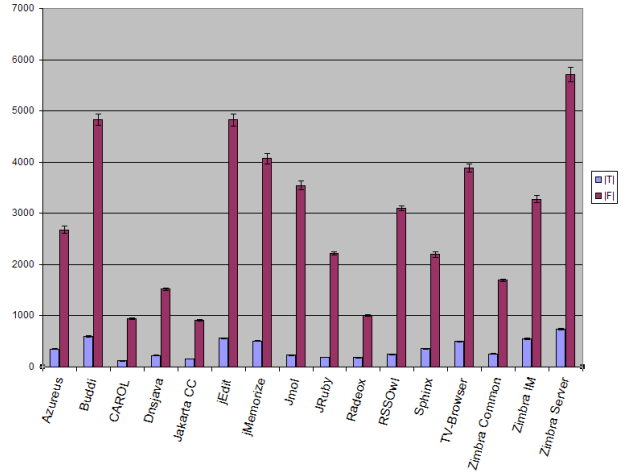
Another factor in accuracy is ambiguity. We recorded the number of ambiguous Function Trees at each test site. We found that 97% of the method calls were unambiguous (meaning there was only 1 type-correct way to fit the functions together). Table 3 shows the accuracy of each algorithm given the number of possible Function Trees. Note that we would expect the algorithms to get at most 50% accuracy when there are two possible trees, since they prefer them each equally.

| Algorithm    | # Possible Trees | Accuracy | Samples |
|--------------|------------------|----------|---------|
| Keyword Tree | 1                | 0.975    | 7796    |
| Bottom-up    | 1                | 0.939    | 7796    |
| Keyword Tree | 2                | 0.434    | 173     |
| Bottom-up    | 2                | 0.272    | 173     |

**Table 3.** Dealing with Ambiguity



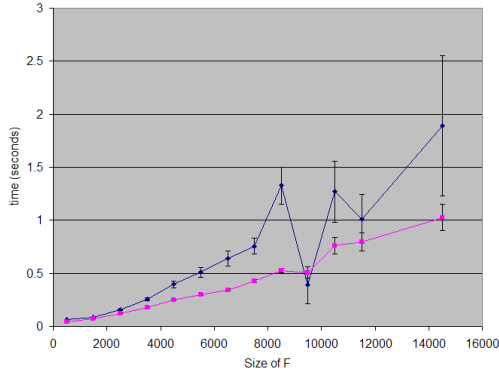
**Figure 3.** Time given # Keywords



**Figure 4.** Size of  $T$  and  $F$  for different projects.

Figure 3 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 0.5 seconds for large inputs. The Keyword Tree algorithm doesn't scale as well to large inputs, taking as long as 2 seconds for each input.

Another factor contributing to running time is the size of  $T$  and  $F$ . Table 4 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 5 shows running time as a function of the size of  $F$ . We see that the Bottom-up algorithm takes about 1 second when  $F$  contains 14000 functions.



**Figure 5.** Time given size of  $F$

One question we have about when the algorithms fail is: are they failing because they can't find a Function Tree with a high score, or because they do find a Function Tree with a high score, but it is the wrong one. If the former is true, then we may need to spend more time searching. If the latter is true, then we may need to think of a better scoring mechanism.

To shed light on this issue for the Bottom-up algorithm, each time it failed to find the correct Function Tree, we recorded the ratio between the score of the Function Tree it did find, and the score of the correct Function Tree. We found the average of this ratio to be 0.964, which is pretty high. This means that the algorithm is usually able to find a Function Tree with a score almost as high as the correct tree.

## 6.2 User Supplied Corpus

## 7. Results

## 8. Discussion

## 9. Conclusions and Future Work

\* Conclusions and Future Work - evaluate a working system on users

### 9.1 Extensions

"However, we believe such features can be seamlessly integrated." - don't say "we believe" or "seamlessly". Just say how they can be integrated. Need to mention using a spell-checker here (approximate matcher) - what about synonyms? Do you handle those?

"We can also incorporate statistical data about functions." - "statistical data" is too vague. Say something like "The system's predictions would be better if it knew the prior probabilities of different functions." Also say how this might be estimated from a large corpus of existing Java code.

- move the whole section 4.3 to later in the paper

Right now, both algorithms rely heavily on matching entire tokens. This doesn't allow for prefix matching, or matching incorrectly spelled tokens. However, we believe such features can be seamlessly integrated. All that would change is the scoring mechanism that says how well a given func-

tion name matches the input sequence. For the iterative algorithm, we would also need to remember which tokens contributed to this match, and how much they contributed. But this should still be possible.

We can also incorporate statistical data about functions. For instance, if the "remove" method of "Vector" is called much more often than the "removeAllElements" method, then the "remove" method could be given some apriori bonus.

## 9.2 Mapping Extensions

We have considered mappings for other constructs. For some of these, we believe 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 second parameter."

### 9.2.1 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 Keyword Commands 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 in this system.

### 9.2.2 Operators:

Operators can map to functions in the natural manner, but we require multiple versions of them 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 we could just have `(double, +, double, double)`, since all the other numeric primitives are subtypes of `double`. However, this wouldn't allow us to add two numbers, and pass the result to a function that requires an `int`.

### 9.2.3 Assignment:

Assignment gets a little more complicated. Say we want to allow the assignment `x = y`, where `x` is an `int` and `y` is a `short`. We could add `(int, =, int, int)`. Unfortunately, this doesn't prevent us 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 we have tried is adding a special set-function for each variable. In this example, we 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)`. Then we would have a function for integer assignment:

(int, =, ref:int, int). We would also make `ref:int` a subtype of `int`, so that we 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.

#### 9.2.4 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.

#### 9.2.5 Other:

We have thought about how we would add constructs like control flow and class declarations. One paradigm we are interested in 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 our system since it is a code completer, and not an actual interpreter.

## A. Appendix Title

This is the text of the appendix, if you need one.

## Acknowledgments

Acknowledgments, if needed.