Emergent, Crowd-scale Programming Practice in the IDE

Ethan Fast[1], **Daniel Steffee**[1], **Lucy Wang**[1], **Joel Brandt**[2], **Michael S. Bernstein**[1]
Stanford University[1], Adobe Research[2]
{ethan.fast, dsteffee, lucywang, msb}@cs.stanford.edu, joel.brandt@adobe.com

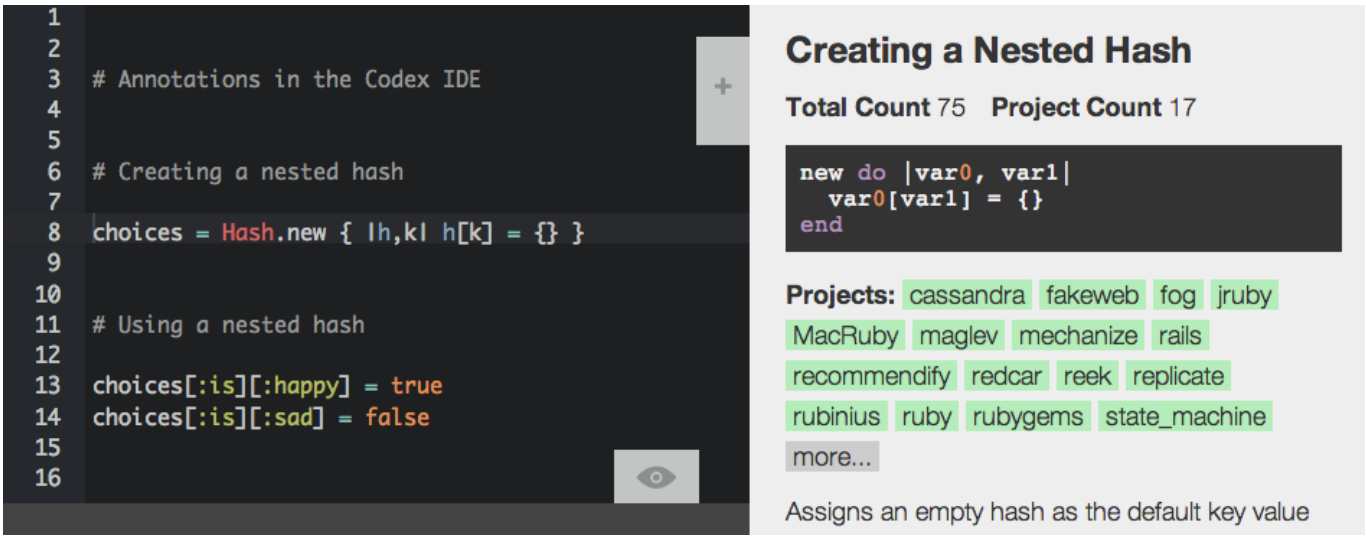

Figure 1. Codex draws on millions of lines of open source code to create software engineering interfaces that integrate emergent programming practice. Here, Codex's pattern annotation calls out popular idioms that appear in the user's code.

ABSTRACT

While emergent behaviors are uncodified across many domains such as programming and writing, interfaces need explicit rules to support users. We hypothesize that by codifying emergent programming behavior, software engineering interfaces can support a far broader set of developer needs. To explore this idea, we built Codex, a knowledge base that records common practice for the Ruby programming language by indexing over three million lines of popular code. Codex enables new data-driven interfaces for programming systems: *statistical linting*, identifying code that is unlikely to occur in practice and may constitute a bug; *pattern annotation*, automatically discovering common programming idioms and annotating them with metadata using expert crowdsourcing; and *library generation*, constructing a utility package that encapsulates and reflects emergent software practice. We evaluate these applications to find Codex captures a broad swatch of programming practice, statistical linting detects problematic code snippets, and pattern annotation discovers nontrivial idioms such as basic HTTP authentication and database migration templates. Our work suggests that operationalizing practice-driven knowledge in structured domains such as programming can enable a new class of user interfaces.

CHI 2014, April 26–May 1, 2014, Toronto, Ontario, Canada.

http://dx.doi.org/10.1145/2556288.2556998

Author Keywords

Programming tools, data mining

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation: Graphical user interfaces

INTRODUCTION

The way people adapt to a system can be just as informative as its original design. In software engineering systems, user practice and designer intention differ across several levels of abstraction: programmers use library APIs in undocumented and unexpected ways [26], language idioms evolve over time [34], and programmers repurpose source code for new tasks [3, 9]. Norms emerge for programming systems that aren't codified in documentation or on the web. What is the best idiom or library to use for a task? Does my code follow common practice? How is a language being used *today*?

We can examine the ecosystem of open source software to find answers to these practice-driven questions. The informal rules and conventions of programming languages and libraries are implicitly present in open source projects, which,

when analyzed, illuminate the ways people code that are too complex or uncommon to appear in official forms of documentation. We can operationalize this knowledge to support everyday programming practice.

Here we present *Codex*, a knowledge base that models practice-driven knowledge for the Ruby programming language. Codex provides a living corpus of how programmers write code, informed by popular open source Ruby projects. The system normalizes program abstract syntax trees (ASTs) to collapse similar idioms and identifiers, filters these idioms and annotates them using paid crowd experts, and then allows applications to query its database in support of new data-driven programming interfaces.

In the domain of programming, emergent practice develops at both the *high-level* of programming idioms, for example code that initializes a nested hash, and at the *low-level* of code syntax, for example blocks that return the result of an addition operation. Codex seeks to capture both higher-level patterns of reuseable program components and lower-level combinations and chains of more basic programming units. Through the *pattern finding* module, Codex identifies commonly reused Ruby idioms. This module uses typicality analysis to identify idioms (e.g., `Hash.new { |h,k| h[k] = {} }`, the most accepted way to initialize a nested hash table). Expert crowds then attach metadata to these idioms, such as a title, description, and measure of recommended usefulness. Alternatively, using the *statistical analysis* module, Codex can compute the frequencies of AST node combinations (e.g., the number of times one function has been chained with another), describing the uniqueness of syntatical patterns.

We present three applications that demonstrate how Codex supports programming practice and software engineering interfaces. First, *pattern annotation* automatically annotates Ruby idioms inside the IDE and presents these annotated snippets through a search interface. Second, *statistical linting* identifies problematic syntax by checking code features (e.g., the kinds of AST nodes used as function signatures or return values) against a large database of trusted and idiomatic snippets; more generally, these statistics give programmers a tool to quantify the uniqueness of their code. Finally, *library generation* pulls particularly common Ruby idioms into a new standard library — authored not by individual developers but by emergent software practice — helping programmers avoid the redefinition of common program components.

This paper makes the following primary contributions:

- Codex: a knowledge base that codifies how developers use programming languages in practice.
- An algorithm for finding common idioms using typicality analysis and expert crowdsourcing; an interface for calling out the idioms and metadata in the Codex IDE.
- *CodexLint*: an approach for *unlikely code* detection that identifies potentially undesirable syntax; an interface for displaying these warnings in the Codex IDE.
- *CodexLib*: An open source Ruby Gem that encapsulates common code patterns into a new library.

Codex enables new software engineering applications that are supported by large-scale programming behavior rather than sets of special-cased rules. While other projects have crowdsourced documentation for existing library functions [28, 8], mined code to enable query-based searching for patterns or examples [26, 33], or embedded example-finding tools into an IDE [5, 7, 15, 29], Codex augments traditional data mining techniques with crowds, presenting a broad data-driven window into programming convention. We demonstrate how these kinds of emergent behavior can inform new design opportunities for user interfaces.

RELATED WORK

Codex draws on techniques from software repository mining to extract patterns from a large body of open source code. Other researchers have mined code for software patterns and redundant code using code normalization or typicality [26, 3, 9, 20, 28, 8]. However, much of this research emphasizes the discovery of known design patterns and is oriented towards applications such as refactoring of duplicate code, while Codex discovers new patterns from the ground up. Further, Codex combines typicality analysis with expert crowdsourcing to build its database — an approach independant of any particular code normalization scheme.

Databases can also systematize knowledge about open source code. However, these databases are usually designed to enable specific forms of code search [35, 33], example-finding [22, 16, 29], or autocompletion [21], either query based or automatic. While tools designed for specific use cases may be highly optimized for their tasks, Codex enables a broader set of applications, including pattern annotation and detecting problematic code through statistical linting.

One of Codex's core applications is to help programmers avoid bugs. Much work has focused on tools for static and dynamic analysis [2, 10]. Other work has focused on helping users debug their programs through program analysis or crowdsourced aggregation of user activities [18, 1, 14, 24, 28]. Codex does not explicitly try to discover bugs in programs; rather, it notifies users when code violates convention. This is a subtle but important difference: code may be syntactically correct but semantically unusual and error-prone.

Codex takes inspiration from prior research on code example finding and reuse. Some of these tools rely on official forms of documentation [5] and others focus on real code from the web [29, 19, 32]. Codex generalizes this work — it covers a broader set of examples than manually curated datasets and can determine when an example is a one-off and when it represents more general practice. Codex also enables a more powerful search over examples through AST analysis, benefits from the human-powered filtering and annotation, and makes possible many applications besides example-finding.

Researchers have also addressed how programmers make use of example code, whether the code is copy-pasted [23] or foraged from documentation or online examples [7, 6, 15]. By formalizing embedded software practice, Codex is able to support programmers through a larger space of examples

and lower-level conventions. Many of these idioms and code snippets may not have been formally discussed on the web.

Codex draws on insights from data-driven interfaces in non-programming domains. Users can gain much through querying and exploration. For example, Webzeigeist allows designers to query a large corpus of rendered web sites [25]. Crowd data also allows interactive systems to transform a partial sketch of the users intent into a complete state, for example matching a sung melody against a large database of music to produce an automatic backup band [31]. Algorithms can then identify patterns in crowd behavior and percolate them up to the interface, for example answering a wide variety of user queries, demonstrating how a given feature is used in practice [4, 11, 27], or predicting likely actions from past history [17]. Codex demonstrates that the more structured nature of programming languages provides a platform for more powerful interactive support such as error finding.

CODEX APPLICATIONS

To ground the opportunities that Codex creates, we begin by introducing three software engineering applications that draw on the Codex data model and high or low level code analysis. In general, Codex enables interfaces and applications that are supported by emergent programming behavior rather than a set of special-cased rules. Following this section, we discuss the techniques behind these applications in more detail.

Statistical Linting

Sometimes, developers program badly: they write code that performs in unexpected ways or violates language conventions. Poorly written code causes significant damage to software projects; bugs tax programmers' time and energy, and code written in an abstruse or non-idiomatic style is more difficult to maintain [30, 13]. Given the complexity of programming languages, rule-based linters can't catch much of this unusual or non-idiomatic code.

Codex operates on the insight that poorly written code is often syntactically different from well written code. For example, functions might be used in the wrong combination or order. So if we collect and index a set of code that is representative of best practices, bad code will often diverge syntactically from the code in this index. Not all syntactically divergent code is bad — the space of well written Ruby programs is very large — but by applying high-precision detectors to a few general AST patterns, Codex can detect syntactically divergent code that is likely to be problematic.

Function Chaining and Composition

Programmers frequently chain and compose functions and operators to create complex algorithmic pipelines, but chaining the wrong kinds of functions together will often cause subtle program bugs. For example, bugs might arise from functions chained in the wrong order, or variables added or assigned in ways they should not be. Codex helps programmers find potential bugs in function chains by identifying unlikely combinations of functions.

For example, if Ava is querying a database that has been normalized to lower case, she needs to convert a string held by the variable `name` to lower case form. She intends to assign the lower case variant of `name` to the variable `lower_case_name` and use this new varible in her query. The Ruby methods `downcase` and `downcase!` will both convert a string variable to lower case, and without thinking too deeply, Ava codes: `lower_case_name = name.downcase!`

Unfortunately, Ava has forgotten that `downcase!` has a side-effect: it changes the variable `name` in place and returns itself. The function she ought to have used is `downcase`, which returns a new lower cased string and does not change `name`. When Ava later uses `name` elsewhere in her program, it doesn't hold the value she expects.

Codex indicates that the line of code is statistically unlikely: `downcase!` is not commonly chained with an assignment statement (although such code is not technically incorrect). Codex notifies Ava that it has observed `downcase!` 57 times, and the abstraction `var = var.`*`any_method`* more than 100,000 times, but it has only encountered one variant of Ava's combined snippet. However, Codex has encountered variants of the correct snippet, `lower_case_name = name.downcase`, more than 200 times.

Block Return Value Analysis

Ruby programmers often manipulate data by passing blocks (lambda-like closures) to functions, but using the wrong kind of block, or passing a block to the wrong kind of function, can process data in unintended ways. Codex identifies unlikey pairings of functions and block return values.

For example, as part of data analysis pipeline, Ash wants to raise every number in a list to the power of 2. He tries to do this with a map block, but encounters a problem (he uses the operator `^` in place of `**`) and adds a `puts` (print) statement inside the map block to help him debug his mistake:

```
new_nums =
    nums.map do |x|
        x^2
        puts x
    end
```

In doing so, Ash has introduced another bug. The `puts` method returns `nil`, which means that `new_nums` will be a list of `nil`. When Ash runs his code, this new error complicates his old problem.

Codex returns a warning: most programmers do not return the method `puts` from a map block. We can anchor this concern in data: Codex has observed `map` blocks 4297 times and `puts` statements 5335 times, but it has never observed `puts` as the last line (an implicit return) of a `map` block. However, Codex observes that `puts` statements are a common return value of blocks that are predominantly used for control flow, like `each` (observed 272 times), so it produces a warning.

Function Type Analysis

Passing the wrong kinds of arguments to a function, or passing positional arguments in the wrong order, can lead to many subtle bugs — especially in duck typed languages like Ruby. However, by analyzing the kinds of AST nodes passed as positional arguments to functions, Codex can warn users about unlikely function signatures.

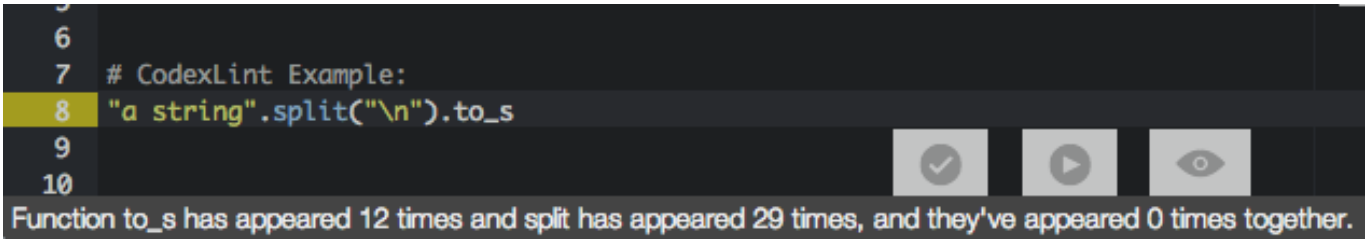

Figure 2. The Codex IDE calls out a snippet of unlikely code by a yellow highlight in its gutter. Warning text appears in the footer.

For example, Severian wants to divide a few hundred datapoints into ten buckets, depending on their id number. To do this, he needs to initialize an array of ten elements, where each element is a hash. Severian codes: `Array.new({},10)`.

Unfortunately, Severian doesn't often initialize arrays with specific lengths and values, and he has reversed the arguments of `Array.new`. When he executes his code, it fails with the error, "TypeError: can't convert Hash into Integer."

Codex could have told Severian that programmers don't often pass `Array.new` an argument list composed of a `string` and `integer`. While Codex observes `Array.new` 674 times, it has never observed `Array.new` with string and integer arguments. However, Codex observes the correct parameterization `Array.new(integer,string)` several times, which is the correct version of Severian's code.

Variable Name Analysis

Good variable names provide important signals about how a variable should be treated and lead to more readable code [30]. Likewise, badly named variables can lead to poor code readability and downstream program errors. By analyzing variable name associations with primitive values (e.g., Strings, Integers, Hashes), Codex can warn programmers about violations of naming conventions.

For example, Azazel is writing a complicated function to process a large dataset from a database call. He is collecting the data in an an `Array` called `array`. However, he later realizes that a hash would be simpler to manage and changes the variable type. In a rush, Azazel changes the variable's type but doesn't bother to change its name: `array = {}`. Later, Ash, who is Azazel's coworker, is looking elsewhere in the function and sees a line `array.keys { ... }`. He wonders, does an `Array` have `keys`? He hadn't thought so.

Instead, Codex notifies Azazel that most programmers do not initialize a variable named `array` with a `Hash` value. While Codex observes initializations to variables named `array` 116 times and variables assigned a `Hash` value many thousands of times, it has never observed the two together. Instead, Codex observes `array = []` 46 times.

It is not wrong to assign a `Hash` value to a variable named `array`, but code that does so is likely less readable and might lead to downstream errors. Codex can determine that such an assignment violates Ruby convention. Likewise, Codex would notice integers stored in `str` or common loop count iterators like `i` being initialized with other variable types.

The Codex IDE integrates CodexLint (Figure 2), allowing users to call up statistics about any line of code in the editor. The linter also runs behind the scenes during development, highlighting any unlikely code with a yellow overlay on the window gutter. When the cursor moves over a marked line, a small message appears on the lower bar of the Codex window, e.g., "We have seen the function split 30,000 times and strip 20,000 times, but we've never seen them chained together."

Pattern Annotation

Many valuable programming idioms are not collected in documentation or on the web. While users can access standard library documentation for core abstractions (e.g., for Ruby, **`http://ruby-doc.org/`**), and libraries often ship with similar kinds of documentation provided by their maintainers, the common idioms by which basic functions may be combined and extended often remain uncodified. Instead, these idioms live in the minds of programmers and — sometimes — on the message boards of communities and forums. Novice users of languages and libraries must "mind the gap" present in official forms of documentation.

Codex fills in gaps of practice-driven knowledge by detecting common idioms as it indexes code and sending them out to be filtered and annotated by a crowd of human experts. Codex finds these idioms by selecting snippets in its database with query parameters such as commonality and complexity. These selected snippets (e.g., that appear in a large number of unique projects and are sufficiently nontrivial) are primed for annotation and human filtering. For instance, over the course of its indexing, Codex identifies `inject { |x,y| x + y }` as a common snippet, occurring 15 times across 4 projects.

Next, Codex sends these snippets — strings of Ruby code, along with examples of them in use — to a Ruby expert on oDesk, a paid expert crowdsourcing platform. The worker annotates the snippet with a title (e.g., "sum all the elements in a list"), a description, and a vote of how useful the snippet would be for an everyday Ruby programmer. Codex stores these annotations in its index along with the original source snippet, making previously implicit knowledge explicit. Eventually, we envision a community of Ruby programmers that annotates snippets of interest.

The Codex IDE uses this annotated snippet information to provide higher-level interpretability to code. The annotations appear whenever a programmer opens a file containing the idiom. Users benefit from annotated code under many different scenarios: perhaps using code scavenged from a web tutorial;

Example Codex Annotated Snippets

HTTP Basic Auth

```
if var0.user
    var1.basic_auth(var0.user, var0.password)
end
```

Sets the basic-auth parameters (username and password) before making an HTTP request, perhaps using Net::HTTP [...]

Popping Options Hash from Arguments

```
if Hash === var0.last
    var0.pop
else
    {}
end
```

Pops the last element from the list 'var0' if it is a Hash. Gives an empty hash if the last element is not a Hash [...]

Raise StandardError

```
raise(StandardError.new("str0"))
```

Raise a StandardError exception using "str0" as exception message [...]

Configure action_controller to disable caching

```
config.action_controller.perform_caching=(false)
```

This will set a global configuration related to caching in action_controller to false [...]

Create a migration template

```
record do |var0|
  var0.migration_template("str0", "str1")
end
```

Create a migration template using "str0" as the source and "str1" as destination. Will create a database-migration [...]

Table 1. Codex identifies common programming snippets automatically, then feeds them to crowdsourced expert programmers for metadata such as the bolded title and descriptive text.

opening an unfamiliar file passed on by a collaborator; revisiting a segment of copy/pasted code; or trying to recall the use of an idiosyncratic library function.

Consider one such user, Morwenna, who is collaborating with a colleague on a Ruby on Rails application. Morwenna hasn't had much experience with Rails, so she begins navigating the many files of her colleague's code in an attempt to build familiarity with the framework. While visiting *config/application.rb*, Morwenna comes across the snippet `config.action_controller.perform_caching = false` and wonders what this means. Codex indicates the line has an annotation, so she asks to see it. The annotation reads, "Turns off default Rails caching."

The Codex IDE calls out and displays any available and relevant annotations (Figure 1). When the cursor moves over a line where annotations are avaiable, a user can call them forward into the sidebar window.

We present examples of these annotated snippets in Table 1. In general, Codex's annotation system uncovers higher-level connections between more basic program components. For instance, human workers can infer the relation of a snippet to some outside library, providing context that isn't explicitly present (e.g., `Net:HTTP` or Ruby on Rails). Similarly, Codex allows for the documentation of higher-level idioms,

Function	Description
`Array#sort_by_index(idx)`	Sort an array by the value at `idx`
`Array#convert_join(str)`	Converts each array element to a string then joins them all on `str`
`Array#upto_size`	Create a range, same size as the array
`String#capital_tokens(str)`	Capitalize all tokens in a string
`Hash.nested`	Create a hash with default value {}
`Hash#get(key)`	Retrieve based on :key or "key"
`File#try_close`	Close a file if it's open

Table 2. A sample of functions from CodexLib, detected in emergent programming practice and encapulated into a new standard library.

where programmers can find each component in documention but not the snippet itself, like the combination of `raise` and `StandardError.new`.

Querying for Understanding

In addition to the automatic idiom detection provided by the *pattern finding* module, users can query Codex directly to better understand community practices around a line or block of code. Queryable parameters include the type of AST node (e.g., a block, conditional, or function call), the body string of the normalized code associated with a node, the body string of original code, the amount of information contained in an AST node (i.e., a measure of code complexity), and the frequency of a node's occurrence across files and projects.

For instance, from a library-driven standpoint, suppose that programmers want to know more about how people use the `Net::HTTP` class. They can query for all blocks that contain `Net::HTTP.new`, sorting on the ones that occur most often. By the diversity of this result set, programmers gain a sense of the kinds of context in which `Net::HTTP` is used — even more so, if any of the results have been annotated by Codex's crowdsourcing engine. This is a more query-driven approach to example-driven development [5, 28].

Queries also have applications in other more IDE-specific components like auto-complete, where the IDE might attempt to find the most common completion for a snippet of code, given additional program context. For example, with the line `Hash.new` and an open block, Codex suggests the completion block `{ |h,k| h[k] = [] }`, which initializes the default value of a `hash` to a new empty `Array`.

Codex's user query system enables a broad set of functionalities including code search, auto-complete, and example discovery. The details of the query language are discussed in the *Codex* section.

Library Generation

Many of the Ruby snippets discovered by Codex are modular, reusable components. The recomposable nature of these snippets suggests that programmers might benefit from their encapsulation in a new standard library that reflects the "missing" functionality that Ruby programmers actually use. Programmers may sometimes engage in unnecessary work: both the mechanical work of typing out repetitive syntax, and also

the mental work of caching task-oriented semantics in working memory.

Here we present *CodexLib*, a library created by emergent practice (Table 2). Unlike human language, which evolves over time (e.g., "personal computer" becomes "PC" and "smartphone" emerges to describe a new class of devices), programming languages and libraries often remain more static. CodexLib suggests programming libraries can similarly evolve based on actual usage.

Consider one common Ruby idiom, creating a new `Hash` object where its default lookup value is another empty `Hash`. This nested hash object allows programmers to write code in a matrix-like style, e.g., `hash[''Gaiman''][''Coraline''] = true`. Programmers usually create a nested hash with the snippet, `Hash.new { |h,k| h[k] = {} }`. The nested hash idiom is 22 characters long and involves some nontrivial tracking of syntactic details, yet it appears in 66 times in 12 projects. Programmers would likely benefit by the creation of a shorter library function. Using CodexLib, they can create a new nested hash with the code `Hash.nested`, which is only 10 characters long and has far fewer opportunities for error.

Alternatively, consider the Ruby idiom to capitalize each word token in a string, which occurs 10 times across 5 different projects:

```
var0.split(/str0/).map do |var1|
    var1.capitalize
end.join("str0")
```

This idiom is dense and not immediately self-descriptive; it contains four function calls and a block within three lines. The code splits `var0` on `str0` (in practice, usually " ") to produce an array, applies `capitalize` to each element in this array, the uses `join` to knit the array into a new string again using `str0`. Programmers might benefit from a simpler way to express this task. Using CodexLib they can achieve the same result with the shorthand code: `var0.capital_tokens(''str0'')`.

CodexLib is a layer on top of the Codex snippet database. To construct it, we extract the most popular idioms and their crowdsourced descriptions from the database. For this small number of functions, it is feasible to manually write function signatures and encapsulate them in new class methods for Hash, Array, String, Float, File, and IO (Table 2). Programmers can download this library as a Ruby gem at **`http://hci.st/codexlib`**.

CODEX

Norms of practice and convention emerge for software systems that aren't codified in documentation. Codex uncovers these norms by processing and aggregating millions of lines of open source code from popular Ruby projects on Github.

Indexing and Abstraction

To build its database, Codex indexes more that 3,000,000 lines of code from 100 popular Ruby projects on Github. It gathers these projects through the Github API by sorting all Ruby projects on the number of watchers and then selecting the 100 projects most watched by other Github users. Codex first breaks apart a project recursively into all constituent AST nodes and annotates these nodes with metadata; next, it normalizes all the AST nodes and collapses those that share a normalized form into a single generalized database entry. The unparsed representation of each of these normalized nodes is a Codex *snippet*.

Snippets of Ruby source code tend to be syntactically unique due to high variance in identifier names and primitive values. Pattern finding tools usually need to abstract away some properties if they are to find meaningful statistical patterns [18, 3, 9]. While we might implement normalization in many different ways, Codex groups together snippets that are functionally similar by standardizing the names of local variables and primatives. For some snippets (e.g., variable assignment) Codex also keeps track of the original identifiers to enable variable name analysis.

Specifically, Codex's normalization renames variable identifiers, strings, symbols, and numbers. The first unique variable in a snippet would be renamed `var0`, the next `var1`, the first string `str0`, and so on. Codex does not normalize class constants and function calls, as these abstractions provide information important to Codex's task-oriented search functionality and statistical linting. As programmers use many different variable names and primitive values when accomplishing a specific task, abstracting away these names helps Codex represent the core behavior of a snippet.

For instance, consider the Ruby snippet:

```
[:CHI, :UIST].map do |z|
    z.to_s + ''is a conference''
end
```

After normalization, this snippet will be:

```
[:sym1, :sym2].map do |var1|
    var1.to_s + ''str1''
end
```

Normalization works less well when such primitives (e.g., specific string or number values) are vital to the interpretation of a snippet. In the future, we will only normalize snippet variable names and identifiers if there is sufficient entropy in their definitions across similar snippets. Snippets with vital identifiers are likely to be more consistent. Other normalization schemes may succeed as well, but we find that this approach successfully collapses most similar snippets together.

Codex applies a map-reduce to the database, collapsing AST nodes with the same normalized form into a single AST entry. We collect additional parameters as part of the map-reduce step: *files*, a list of each file in which the snippet occurs; *projects*, a list of projects in which the snippets appears; *count*, the total number of times a snippet has appeared; *file_count*, the number of times a snippet has appeard in unique files; and *project_count*, the number of times a snippet has appeared in unique projects. Codex uses these parameters to enable the statistical and pattern finding modules.

Codex users the Parser and AST Ruby gems by *whitequark* for AST processing. We have deployed the Codex database on Heroku, using RethinkDB and MongoHQ.

Statistical Analysis Module

Codex has two modules that together enable both *high-level* and *low-level* pattern detection. First we describe the low-level module, which focuses on syntactical patterns that occur among AST nodes.

The statistical analysis module allows Codex to warn users when code is *unlikely*. Codex decides this likelihood using a set of statistics: the frequency of the snippet and also the frequencies of component forms of the snippet (e.g., `.to_s` and `.split` for `.split.to_s`). When a snippet's component forms are sufficiently common and the snippet itself is sufficienctly uncommon, Codex labels it unlikely; that is, a snippet s must have occurred fewer than t times and all its component pieces, c_i must have occurred at most t_i times.

Detecting Surprisingly Unlikely Code

Codex indexes many kinds of AST nodes (e.g., blocks, conditionals, assignment statements, function calls, function definitions), but it conducts syntactic analysis upon a subset of these nodes. The function by which a snippet of unlikely code is declared surprising differs based upon the type of node in question. We discuss four representative analyses we have built to demonstrate the system's power:

1. *Function Call Analysis*: This analysis checks how many times a function has been called with a given "type signature", which Codex defines as the kind of AST nodes passed as arguments (not the runtime type of the expression), relative to the number of times the function has been called with other kinds of signatures. If a sufficiently common function appears with a type signature that is very rarely observed by Codex, this may suggest problematic code. In `split(' ',2)`, s is `split(string,number)`; c_1 is the name of the function; c_2 is the function signature, e.g., `[string, number]`). Codex checks how many times `split` is called with string and integer arguments relative to other kinds of arguments.

2. *Function Chaining Analysis*: This analysis checks how many times one function has been chained with another; that is, the result of some first function is used as the caller of some second function. Here s is the function chain, e.g., `split.to_s`; c_1 is the first function, e.g, `split`; and c_2 is the second, e.g., `to_s`. Two functions that are often used but never chained together suggest unusual code.

3. *Block Return Value Analysis*: This analysis checks how many times a certain kind of block has returned a certain kind of value. For instance, it would be legal but unusual to write the code `things.each { |x| x.to_s }`, which does transform every element in the `things` list to a string, but does not alter `things` itself since `to_s` does not change the state of its caller (to change the values in `names`, a programmer might instead use the expression `x = x.to_s` inside the `each` block). Here s is a kind of block with a particular return type, e.g., a `each` block with return type of the `to_s` function; c_1 is a kind of block, e.g., an `each` block; and c_2 is a kind of block return type, e.g., blocks returning the `to_s` function.

4. *Identifier Analysis*: This analysis checks how many times a variable identifier has been assigned with a certain type of primitive. Often variable names suggest the type of the variable that they reference; this analysis allows Codex to warn programmers about misleading or unconventional variable names (e.g., `str = 0` or `my_array = {}`). Here s is the variable name as assigned to a particular type, e.g., `str = 0`; and c_1 is the variable name, e.g., `str`.

Pattern Finding Module

Whereas the statistical analysis module focuses on low-level syntactical structure, the pattern finding module detects a set of high-level Ruby idioms and example snippets commonly reused by programmers. By constructing an appropriate query over the normalized snippets in its database, Codex can find snippets that isolate common programming idioms. The pattern finding module also enables other specific kinds of queries based on context (e.g., searching for certain library methods called from within a map block.)

The general form of Codex's pattern finding consists of a single query that is applied to the database of abstracted snippets; we intend it to filter out snippets that programmers are less likely to find interesting or useful. The query has five parameters, corresponding to attributes stored in the database, and ordered here by their selectivity:

1. *Project Count*: the number of unique projects in which an abstracted snippet has occurred. A lower bound of 2% of the number of projects indexed by codex filters out snippets that tend to be longer and more idiosyncratic.

2. *Total Count*: the total number of times an abstracted snippet has occurred. An upper bound of the 90% percentile filters out overly trivial snippets (e.g., `var0 = var1`).

3. *File Count*: the total number of unique files in an abstracted snippet has occurred. An upper bound of 20% of the count of an abstracted snippet filters out snippets that are reused quite a bit within one or more files; these snippets tend to be overly domain specific.

4. *Token Count*: the number of unique variables, function calls, and primitives that occur in an abstracted snippet. An upper bound of the 80% percentile of all snippet token counts filters out overly domain specific code.

5. *Function Count*: the number of unique function calls in a snippet. A lower bound of 2 filters out trivial snippets.

These snippets are then passed to expert crowds, who attach metadata such as a title, description, and measure of recommended usefulness.

Together, these parameters produce 9,693 abstracted snippets from the Codex database, corresponding to 79,720 original snippets in the index. This query is designed to produce general purpose snippets; other queries might be constructed differently to produce more domain specific results.

EVALUATION

Codex hypothesizes that we can build new software engineering interfaces by using databases that model practice-driven

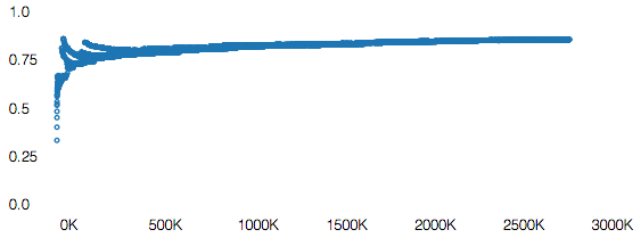

Figure 3. A plot of Codex's hit rate as it indexes code over four random samples of file orderings. The y-axis plots the database hit rate, and the x-axis plots the number of lines of code indexed.

knowedge for programming languages. In this section, we provide evidence for three claims:

1. *The 3,000,000 snippets in the Codex database are sufficient to characterize and analyze a broad swath of program behavior.* We measure the redundancy of AST nodes as Codex indexes increasing amounts of code.

2. *Codex captures a set of snippets that are recomposable and task-oriented.* We ask oDesk Ruby experts to describe and review a subset of the Codex patterns.

3. *Codex allows us to identify unlikely code, without too many false positives.* We evaluate the number and kinds of warnings that Codex throws across a test set of 49,735 lines of code.

The Codex Database

The Codex database is composed of more than 3,000,000 lines of open source code, indexed from 100 popular Ruby projects on Github. These projects come from a diverse set of application areas, including programming languages, plugins, webservers, web applications, databases, testing suites, and API wrappers.

We designed Codex to reflect programming practice. Programming is open ended — the number of valid strings of source code in most languages is infinite — so no database can hold information about every possible AST node or program. However, programming is also highly redundant when examined at a small enough level of granularity [12]. Of the approximately 7 million AST nodes that Codex has indexed, only 13% are unique after normalization. Among the more complex types of AST nodes we see variablity in this redundance. For example, among block nodes 74% are unique, and among class nodes 85% are unique (Table 3).

To evaluate the breadth of code that Codex knows about, we examine the overall *hit rate* of its database as it indexes more code. That is, when indexing N lines of code, what percentage of its normalized AST nodes have not been seen before as they are added to the database? We analyzed the raw Codex dataset for values ranging from 92 to 3,000,000 lines of code across four random samples of file ordering.

Codex's hit rate exceeds 80% after 500,000 lines of code (Figure 3), meaning that Codex had already observed 80% of the AST nodes after normalization. Different AST node types display slightly different curves, with the same overall shape.

Node Type	Percent Unique
Class definition	85%
Rescue statement	78%
Block statement	74%
Function definition	69%
If statement	66%
Interpolated string	29%
Function call	28%
Inlined hash	17%

Table 3. The percent of snippets that are unique after normalization for common AST node types.

Category	Percent of Snippets
Standard Library	76%
External Library	14%
Data or Control Flow	9%

Table 4. Programmers from an expert crowdsourcing market annotated Codex's idioms with their usage type. The vast majority concern the use of standard, built-in libraries.

Many of the nodes we are interested in for statistical analysis are more complex, and so they are less amenable to the leveling of this curve. However, were Codex to index more code, its hit rate would increase even futher.

Pattern Annotation

We asked professional Ruby programmers on the oDesk expert crowdsourcing marketplace to annotate 500 Codex snippets randomly sampled from the approximately 10,000 snippets that passed Codex's general pattern finding filter.

First, we asked crowdworkers to label each snippet with one of the categories: Data or Control Flow, Standard library, External library, and Other (Table 4). The majority of snippets address standard library tasks (76%), followed by external library tasks (14%), and tasks involving data or control flow (9%). None fell outside these categories (Other = 0%).

Next, we asked oDesk crowdworkers to answer: 1) Is this snippet a useful programming task or idiom? 2) Can this snippet be encapsulated into a seperate standalone function? 3) Is there a more common way to write this snippet?

The oDesk Ruby experts reported that 86% of the snippets queued for annotation are useful, 96% are recomposable, and 91% have no more common form. These statistics indicate that Codex's pattern finding module produces snippets that are generally recomposable and reflective of good programming practice.

Statistical Linting

Stastical linting relies upon the low-level properties of millions of lines of code to warn users about code that is unlikely. Codex defines a general approach for detecting unlikely code, on which it implements analyses for: type signatures, variable names, function chains, and block return types. Here we evaluate to what extent CodexLint's produces *false positives* through a training set of 49,735 lines of code.

As Codex seeks to identify unlikely code, and not program bugs, the distinction between true positives and false positives is largely subjective. Inevitably, some users will want to be warned about these properties, while others will not. Here we test the statistical linter against code known to be of high quality. Supposing the number of warnings CodexLint suggests is small, relative to the number of lines of code analyzed, this provides evidence that the statistical linting tool does not suggest too many false positives.

We based our CodexLint test set on 6 projects randomly sampled and witheld from the 100 repositories collected to build Codex's index. The test set projects contain a total of 49,735 lines of code, and all of these projects are popular and widely used, with more than 100 watchers on Gitub (as the case for all the projects selected for indexing by Codex). Since 90% of the snippets annotated through Codex's pattern finding module are found by crowdsourced experts to be idiomatic, and over 85% are rated as useful, we can safely assume that these projects generally *do* contain high-quality code — the null hypothesis would be the principle, "garbage in, garbage out." By treating each warning it throws as a false positive, we arrive at a conservative estimate of the error rate.

Running CodexLint against the test set, we find that it generates 1248 warnings over 49,735 lines of code; this suggests a conservate false positive rate of 2.5%.

The most common category of false positive involves functions and blocks that appear at least a few times across a number of projects, but that haven't been observed enough for Codex to appropriately model their behavior. For example, `nodes` and `uri` are part of a HTML parsing library that Codex has only seen used in a few files, and the system throws a warning about their combination, e.g., `nodes.uri`. We are working on a new technique to detect sparse functions based on library dependancies and additional program context that will handle them seperately in analysis.

The second most common false positive occurs when Codex observes two AST nodes, neither of them particularly uncommon, together in a new and valid way, e.g., `lambda` blocks returning a function call to `rand`, which did not appear at all in Codex's index. Programming is an open-ended task, and there will always be valid combinations of expressions that a system like Codex has not encountered.

Other false positives are more ambiguous. For example, one project passes the `map` function a string, which would usually produce an error. This project had overridden `map` to support new functionality. Similarly, another file assigns a variable named `@requests` an integer value, and Codex has only ever observed `@requests` as an array. Programmers might be well served by changing their code in response to these warnings.

Finally, this false positive rate will decrease as the size of Codex's index grows and fewer correct code paths surprise it. As the statistical linting algorithm is based upon probability thresholds, users can make the linter even more conservative by adjusting these thresholds — analogous to adjusting the parameters of traditional linters.

LIMITATIONS AND FUTURE WORK

The approach that Codex takes has limitations, many of which we plan to address with future work. First, while we have collected evidence that suggests Codex's index is large enough to encompass a broad swath of program behavior, it is likely that many applications — such as pattern annotation and statistical linting — would benefit from a larger index of code. We have tested Codex with indexes as large as ten million lines of code, with no significant difference in the kinds of nodes and statisitical properties it detects. However, as the size of the index grows, there will be fewer and fewer edge cases and false positives, and Codex will more easily detect idioms and make precise statistical statements about combinations of AST nodes. Codex must balance its desire for more coverage against the danger of indexing lower-quality code.

Second, many more kinds of program analyses can be defined beyond Codex's current abstractions. All the analyses tested in the current version of Codex rely upon local properties of AST nodes, and not the sourrounding program context. By incorporating more of this context into analyses, we might detect more complex properties (e.g., detecting that a user hasn't initialized a MySQL database wrapper).

Third, due to the subjective nature of CodexLint's warnings, we have not determined a precise rate of true positives and false positives. In future work, we might ask programmers to evaluate these warnings, to better determine how often they are useful. Moreover, this paper does not address the general question: do programmers really find it useful to know when they are violating convention? We can determine the answer more concretely through longitudinal study.

Finally, while Codex models practice-driven knowledge for the Ruby programming language, our techniques for processing AST nodes and generating statistics are applicable to any AST structure or language. For example, it might be feasible to generate a Codex database for JavaScript by crawling highly-trafficked web pages. Moreover, while we focused on a dynamic language due to its popularity and flexibility of naturalistic usage, static languages provide additional metadata that Codex could leverage. Extending Codex's analyses to these other languages remains future work.

CONCLUSION

Codex suggests that mining and codifying emergent programmer behavior can support a broad set of software engineering interfaces. By modeling how developers use programming languages in practice, Codex enables algorithms for finding common idioms and detecting unlikely code. In combination with human crowds, we use these algorithms to enable new applications like statistical linting, pattern annotation, and library generation.

More broadly, systems like Codex point towards a future of programming languages as living artifacts: where libraries self-update to use the latest, most common idioms, IDEs offer suggestions to programmers that suit evolving coding styles, and languages evolve to better support their users. In this way, the wisdom of the crowd can be fed back to make crowds themselves wiser.

ACKNOWLEDGEMENTS

Special thanks to our reviewers and colleagues at Stanford. This work is supported by Adobe Research.

REFERENCES

1. Ahmadzadeh, M., Elliman, D., and Higgins, C. An analysis of patterns of debugging among novice computer science students. *In Proc. ITiCSE 2005*.
2. Ayewah, N., et al. Using static analysis to find bugs. *In IEEE Software 2008*.
3. Baxter, I.D., et al. Clone detection using abstract syntax trees. *In Proc. ICSM 1998*.
4. Bernstein, M.S., et al. Direct answers for search queries in the long tail. *In Proc. CHI 2012*.
5. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. Example-centric programming: integrating web search into the development environment. *In Proc. CHI 2010*.
6. Brandt, J., et al. Opportunistic programming: Writing code to prototype, ideate, and discover. *In IEEE Software 2009*.
7. Brandt, J., et al. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *In Proc. CHI 2009*.
8. Buse, R.P.L. and Weimer, W. Synthesizing api usage examples. *In Proc. ICSE 2012*.
9. Ducasse, S., Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. *In Proc. ICSM 1999*.
10. Engler, D., et al. Bugs as deviant behavior: a general approach to inferring errors in systems code. *In Proc. SOSP 2001*.
11. Fourney, A., Mann, R., and Terry, M. Query-feature graphs: bridging user vocabulary and system functionality. *In Proc. UIST 2011*.
12. Gabel, M. and Su, Z. A study of the uniqueness of source code. *In Proc. FSE 2010*.
13. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.
14. Goldman, M., Little, G., and Miller, R.C. Collabode: collaborative coding in the browser. *In Proc. CHASE 2011*.
15. Goldman, M. and Miller, R.C. Codetrail: Connecting source code and web resources. *In Proc. VL/HCC 2009J*.
16. Grechanik, M., et al. Exemplar: Executable examples archive. *In Proc. ICSE 2010*.
17. Greenberg, S. and Witten, I.H. How users repeat their actions on computers: Principles for design of history mechanisms. *In Proc. CHI '88*.
18. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S.R. What would other programmers do: suggesting solutions to error messages. *In Proc. of CHI 2010*.
19. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. Programming by a sample: rapidly creating web applications with d.mix. *In Proc. UIST 2007*.
20. Hindle, A., et al. On the naturalness of software. *In Proc. ICSE 2012*.
21. Holmes, R., Walker, R.J., and Murphy, G.C. Strathcona example recommendation tool. *In Proc. FSE 2005*.
22. Hummel, O., Janjic, W., and Atkinson, C. Code conjurer: Pulling reusable software out of thin air. *IEEE Software 2008*.
23. Kim, M., Bergman, L., Lau, T., and Notkin, D. An ethnographic study of copy and paste programming practices in oopl. *In Proc. ISESE 2004*.
24. Ko, A.J. and Myers, B.A. Designing the whyline: a debugging interface for asking questions about program behavior. *In Proc. the CHI 2004*.
25. Kumar, R., et al. Webzeitgeist: Design Mining the Web. *In Proc. CHI 2013*.
26. Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. Jungloid mining: helping to navigate the api jungle. *In Proc. PLDI 2005*.
27. Matejka, J., Li, W., Grossman, T., and Fitzmaurice, G. CommunityCommands. *In Proc. UIST 2009*.
28. Mooty, M., Faulring, A., Stylos, J., and Myers, B.A. Calcite: Completing code completion for constructors using crowds. *In Proc. VL/HCC 2010*.
29. Sahavechaphan, N. and Claypool, K. Xsnippet: mining for sample code. *In Proc. OOPSLA 2006*.
30. Seacord, R.C., Plakosh, D., and Lewis, G.A. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. 2003.
31. Simon, I., Morris, D., and Basu, S. MySong: automatic accompaniment generation for vocal melodies. *In Proc. CHI 2008*.
32. Stylos, J. and Myers, B.A. Mica: A web-search tool for finding api components and examples. *In Proc. VL/HCC 2006*.
33. Thummalapenta, S. and Xie, T. Parseweb: a programmer assistant for reusing open source code on the web. *In Proc. ASE 2007*.
34. Urma, R.G. and Mycroft, A. Programming language evolution via source code query languages. *In Proc. PLATEAU 2012*.
35. Ye, Y. and Fischer, G. Supporting reuse by delivering task-relevant and personalized information. *In Proc. ICSE 2002*.