Codex: a Metadata Format for Rich Code Exploration

Jonathan Dönszelmann
J.B.Donszelmann@student.tudelft.nl
Delft University of Technology
The Netherlands

Daniël A. A. Pelsmaeker
D.A.A.Pelsmaeker@tudelft.nl
Delft University of Technology
The Netherlands

Danny M. Groenewegen
D.M.Groenewegen@tudelft.nl
Delft University of Technology
The Netherlands

Abstract

Programmers spend significantly more time trying to comprehend existing code than writing new code. They gain an understanding of the code by navigating the code base in an IDE, by reading documentation online, and by browsing code repositories on websites such as GitHub. Creating rich experiences for a variety of programming languages across those various media is a large effort. This effort might be worthwhile for popular languages, but for niche or experimental languages the required effort is often too large. To reduce this effort, we introduce the *Codex metadata format*, separating the language-specific generation of code metadata from its language-agnostic presentation. We demonstrate this approach by implementing four language-specific generators (from LSP, CTAGS, TextMate, and Elaine) and two language-agnostic presentations (this paper, and a code viewer website) of code and metadata. To demonstrate different kinds of code metadata, we implemented four different code exploration services: syntax coloring, code navigation, structure outline, and diagnostic messages. We show that the Codex format successfully decouples languages and their metadata from their presentations.

CCS Concepts: • **Software and its engineering** → **System description languages**; • **Information systems** → **Data exchange**; **Data structures**.

ACM Reference Format:
Jonathan Dönszelmann, Daniël A. A. Pelsmaeker, and Danny M. Groenewegen. 2023. Codex: a Metadata Format for Rich Code Exploration. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (SLE 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

SLE 2023, October 22–27, 2023, Cascais, Lisbon
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As programmers, we spend much more time reading code than writing it. We try to get acquainted with the code base as part of our new job, work our way though the API documentation of the new cutting-edge framework that everyone uses, or attempt to comprehend code written decades ago. All in all, compared to writing code, we spend an estimated ten times more time reading code [12].

We read code in our highly interactive code editors and Integrated Development Environments (IDEs), but also explore code repositories on GitHub, browse API documentation and specifications, look for answers to our questions on StackOverflow, and even internalize code from offline paper-based publications.

We call this *code exploration*: the process of analyzing and understanding a software code base by examining its structure, components, and dependencies. To explore code effectively, we use *code exploration services*, that form a subset of the more well-known *editor services*, but exclude services that are only useful when writing code. Code exploration services range from simple yet effective syntax coloring, all the way to interactive services such as code navigation and hover information. These services are not only useful in a code editor, but also in other media where code might be explored, such as on documentation websites and in books. The code exploration services and their presentations will need to be adjusted to meet the limitations of the specific medium. For example, syntax coloring is feasible in a printed book, but one cannot instantly jump to a definition.

There are existing solutions that facilitate code exploration. Most major programming languages have spent effort to implement their own editor plugins that provide syntax coloring and additional editor services. To support authors of code libraries, more and more languages include tooling to generate language-specific documentation websites from a code base. For example, Rust has RustDoc, Haskell has Haddoc, and Agda can output code as rich HTML and even LaTeX code. However, all these solutions are very narrowly applicable to only particular languages where the developers have gone through the trouble of providing, implementing, and maintaining these services. Especially for languages with less development power, investing in the implementation of such services may not be worthwhile.

At the same time, there are tools that apply to multiple languages but target only a very specific code exploration service. For example, there is Bootlin's Elixir cross referencer, which allows users to navigate C and C++ source code online, such as the Linux Kernel source code[1]. To provide code navigation in editors that support it, CTAGS [19] is a tool that generates an index of identifiers found in a code base and which supports more than a hundred languages.

We will call these *narrow* tools: those that either provide few services for many languages, or many services for few languages. This hints at a problem often referred to as the *IDE portability problem*, where m languages providing services for n editors require $m \times n$ implementations and related effort [10].

There are some attempts to solve the $m \times n$ problem, many of which are experimental. We will discuss some of those in section 6. However, there is one important solution made for editors which is worth mentioning here: the Language Server Protocol [14].

1.1 Language Server Protocol

A somewhat recent development for portable editor services has been the Language Server Protocol (LSP). LSP is a communication protocol between a language server and an editor client. The protocol allows languages and editors to be decoupled, removing the need to spend $m \times n$ effort to support m languages in n editors, while instead making it an $m + n$ problem. This allows m languages to implement a language server, while needing to know nothing about the editor it runs in. Similarly, for n editors to support LSP, they need to know very little about the language they provide editor services for.

LSP inherently needs to be very flexible. Because it decouples editors from languages, neither party can assume much about the other side. At startup, LSP requires negotiation between the two parties about their capabilities. What capabilities can the language server provide, and which can the editor display?

The fact that both many editors and many languages support the protocol[2] makes editor services truly portable. Due to its native support in Microsoft's VS Code, one of the most popular editors, LSP gained a lot of attention and is broadly implemented [18].

As a protocol, LSP relies on a request-response system between an editor client and a language server. This allows LSP to react dynamically to changes to the code base. However, this also makes it difficult to adapt LSP to work outside an editor in other code exploration media. On a website, requests can be made through JavaScript. There is some precedent of tools doing this, like Eclipse Che and Theia, but an inherent delay exists because of the network [4, 5]. Some PDF documents can technically support the execution of code, but this is often blocked by PDF viewers for security reasons. And doing anything dynamic in a paper medium such as a book is simply out of the question.

All the above disadvantages can be mitigated by not requesting the metadata about a code base dynamically, but storing it statically in a language-agnostic data format. So what would such a format look like?

1.2 Codex

In this paper, we present *Codex*: an intermediate format for describing code base metadata, a solution to the $m \times n$ problem for rich code exploration. The format is language-agnostic and can be extended to support new kinds of metadata for future code exploration services. Thanks to its offline nature, the format allows the code to be explored at a point later in time from when the metadata is generated, even when the specific versions of tooling that were used on the code base are no longer available.

We implemented four prototype *generators* of the Codex format, which show how various sources and different techniques can be used to generate metadata describing a code base; and implemented two different rich prototype *presentations* of the Codex format for exploring said code base.

Contributions. In summary, our contributions are:

- the Codex metadata format, a tractable solution to the $m \times n$ problem for code exploration services, suitable for describing code bases of both DSLs and mainstream languages;
- a proof of concept comprising four prototype generators of Codex metadata (LSP, CTAGS, TextMate, Elaine), and two prototype code presentations derived from Codex metadata (LaTeX, HTML);
- an analysis of the trade-offs when designing a code metadata format.

We give a high-level explanation of the Codex format in section 2, and demonstrate a proof of concept in section 3 to show that our idea can indeed power such rich code exploration services. This demonstration is followed by an extensive discussion in section 4 of all the considerations that went into this proof of concept, followed by an analysis of its limitations and how we envision its future in section 5. We finish with a comparison of our solution to related work in section 6, and end with our conclusion in section 7.

2 The Codex Metadata Format

The goal of the Codex format is to disconnect the language-specific generation of code metadata from the language-agnostic presentation of code, as demonstrated in section 3. A language-agnostic presentation implies that the format should be able to describe metadata for any programming

[1]Explore the Linux Kernel source code at https://elixir.bootlin.com
[2]An overview of LSP-supporting tools can be found at https://microsoft.github.io/language-server-protocol/implementors/servers/ and https://microsoft.github.io/language-server-protocol/implementors/tools/

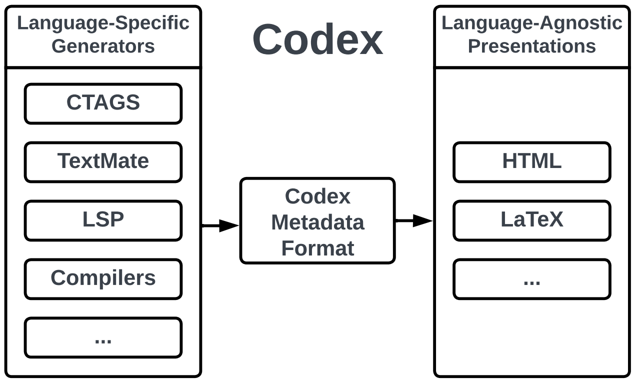


Figure 1. A diagram of Codex. Many language-specific, or *narrow*, generators connect, through the Codex format, to many language-agnostic presentations of a code base.

language, sometimes even multiple, and cannot make assumptions that are only true for some languages.

We consider some aspects of such a metadata format out of scope: the speed at which the metadata is generated, and similarly the speed at which it is consumed to produce a presentation. This is because these are highly dependent on the implementations that generate and use the metadata. As a result, we consider these aspects to be future work.

2.1 Types of Code Exploration Services and Metadata

An important attribute of the Codex static data format is that it not only provides separation of concerns between languages and their viewing media, but also a separation in time. The metadata can be generated at one point in time, and might even be checked into a version control system. Later, any presentation of the code base can be produced by simply querying its metadata and displaying it to the reader.

Metadata generators might not have the capability to generate certain metadata, and similarly, not all metadata might be displayable to the reader in all possible presentations. Therefore, the format is designed such that generators can omit metadata as required, and the presentation should be able to deal with such missing metadata.

The metadata format describes the metadata for a code base for various code exploration services. Erdweg et al. [6] created a feature model of language workbenches and provided a categorization of editor services. From this list and the editor services that LSP supports [14], we derived a list of eight code exploration services which are also useful in static read-only environments outside of editors:

- syntax coloring;
- code folding;
- structure outline;
- code navigation (quick navigation between usages and definitions);
- documentation;
- signature help (hints of type signatures);
- (certain[3]) code actions; and
- diagnostic messages.

To store the metadata for these code exploration services, our data format has the ability to:

- reference specific sections of the source code, which aids code navigation and relating structures in the outline;
- classify certain parts of source code, for example to indicate the syntactic classification of a token or the semantics of a structure;
- add extra information to pieces of source code, such as a diagnostic message from the compiler.

Some code exploration services need multiple, or even all of these kinds of metadata to be present. For example, code navigation mainly and most obviously consists of reference information. However, we also classify what kind of reference a reference is. It could be a reference from a usage to a definition, or the other way around.

2.2 Definition of the Codex Format

In fig. 2a we show the structure of Codex metadata. Our implementation of the Codex format is able to contain metadata for four representative code exploration services out of the eight shown in section 2.1. The implementation of the other four will look similar to the ones shown.

Central to the format is the `Relation` type. In section 2.1 we gave a list of eight code exploration services. The `Relation` type quite literally reflects the metadata required for four of these eight, which are the ones we have implemented so far. Each piece of metadata is represented by referring to a specific stretch of code in a file, which we call a `Span`, which is associated with a certain relation.

These relations internally use several data types to represent metadata. Those data types are based on the three properties we identified should be supported in section 2.1. One of them is, once again, `Span`, to allow metadata to refer to another location in one of the source files. This is used, for example, for code navigation.

In the metadata for the structure outline, the `Span` is used to refer to the enclosing structure. For example to indicate that a field is enclosed in a `struct`, linking the child to the parent.

Next, there is `Classification`, which is used in many relations, but is easiest to explain in the context of syntax coloring. A classification always comes in the form of a sequence of classifiers, ordered from very general to very specific. For

[3]The category of Code actions (as defined for example in LSP, https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_codeAction) is very broad. For example, showing the expansion of a macro in a language like Rust is perfectly viable in a static context, but an automatic refactoring is not, since it modifies the source code.

```rust
// Types
struct Span {
    start: usize,
    length: usize,
    file: Path,
}
struct Classification(Vec<String>);
struct Text(String);

// Relations
enum Relation {
    Outline {
        kind: Classification,
        parent: Span,
    },
    Syntax {
        kind: Classification,
    },
    Reference {
        target: Span,
        kind: Classification,
    },
    Diagnostics {
        kind: Classification,
        message: Text,
    }
}

// Metadata
type Metadata = Vec<(Span, Relation)>;
```

(a) The definition of the Codex metadata format in the Rust proof of concept. In Rust, Enums are sum types, they can hold one of a number of possible variants. For example, a `Relation` can be a `Reference`, or a `Syntax` classification. This is also an example of code formatted using Codex, our proof of concept which we show in section 3.

```
["data_structures.rs!17+14", {
  "Reference":
  {
    "kind": ["declaration"],
    "reference": "data_structures.rs!1+44"
  }
}]
//...
["data_structures.rs!107+4", {
  "Diagnostics":
  {
    "severity": ["error"],
    "message": "struct Path not in scope"
  }
}]
//...
["data_structures.rs!17+14", {
  "Outline":
  {
    "kind": ["struct"],
    "parent": null
  }
}]
//...
[ "data_structures.rs!10+6", {
  "Syntax":
  {
    "kind": ["keyword","declaration","struct","rust"]

  }
}]
```

(b) An excerpt of serialized metadata in the codex format, showing what data for the different kinds of metadata categories defined in section 2.1 look like. The metadata was generated by taking the source code in fig. 2a and running it through the Codex tool we show in section 3. Out of about a hundred lines of metadata, only a few lines are shown.

Figure 2. The Codex format as Rust types and serialized to a JSON-like format.

example, to classify a syntactical token like the Rust keyword `enum`, we use a classification of `keyword.declaration.enum.rust`, where the most generic classifier is `keyword`.

The theme for an editor or a presentation can match on this classification to find the most suitable style. For example, the theme might indicate that all `variables` are rendered in blue, but specifically `variable.parameters` are green. Similarly, a reference might be a usage, `definition` or `implementation`, where an `implementation` might be considered a kind of `usage` and a presentation can treat these differently. The big advantage of such classifications is that if a presentation does not know how to display a certain specific classification, it can fall back to a more generic one. For syntax coloring, this system is quite similar to CSS classes, where specific rules override more generic ones.

This system is inspired by *scopes*[4] of the TextMate editor [11]. They use hierarchical classifications for syntax coloring, of which we also reuse their somewhat standardized classifications.

Finally, there is a need for free-form text in some relations. In fig. 2a, only `Diagnostics` needs that, to include the diagnostic text itself, so the message can be presented with the code.

A small example of what some metadata might look like when using this format can be found in fig. 2b. The example shows samples of each kind of metadata generated from a Rust file in a serialized representation as JSON. The coloring and interactive features of both fig. 2a and fig. 2b are generated using the Codex proof of concept, demonstrated

[4]See: https://macromates.com/manual/en/language_grammars

in section 3. That means that when reading this paper digitally, all underlined parts of fig. 2a are clickable hyperlinks to definitions or usages of identifiers.

2.3 Extensibility

As we cannot predict all the current and future code exploration services for all possible languages, the Codex metadata format, as shown in fig. 2a, is designed to be extensible. The main extension point is to add a new `Relation` with metadata fields for some code exploration service. Those fields can reuse the data types already defined, but could also use some custom data types if necessary.

3 Demonstration

To show that a metadata format can indeed power code exploration services, we have created a proof-of-concept[5]. The goal of this proof of concept is twofold: to demonstrate that it is feasible to use statically generated metadata to provide code exploration services in various media; and to show that by using a data format like ours, we can decouple the production and consumption of such static metadata in a language-agnostic way.

To demonstrate these two goals, we first use metadata in the Codex format to provide rich code exploration on websites and PDF documents. Then, we show two approaches to gather metadata from code bases: adapting existing tools made to support existing languages; and to directly gathering metadata from a parser and type checker of a Domain Specific Language (DSL). This shows how the Codex format is useful to present code for small programming languages for which no custom tooling is available.

3.1 Presenting Code on Websites through HTML

HTML is a standard format for documents designed to be displayed in web browsers. Together with CSS and JavaScript, HTML can be used to create websites with complex graphical user interfaces and visualizations. Many websites such as StackOverflow, GitHub and GitLab can visualize users' source code, sometimes with some basic code exploration services.

In our demonstration we can produce HTML documents from source code and its corresponding Codex metadata. A screenshot of the resulting website is shown in fig. 3a.

In the generated HTML document, we implemented several code exploration services: syntax coloring, structure outline, code navigation, and diagnostic messages. Code navigation is implemented by adding hyperlinks to the code, though when more than one reference exists, a pop-up is shown when hovering over underlined items, allowing users to choose where to navigate. Diagnostic messages are shown by underlining an item in yellow, which when hovered over shows the associated compiler warnings and errors.

It is possible for users to change the theme of the HTML visualization. Instead of assigning colors to tokens, we add CSS classes based on the token's classification (See section 3.3 for more details). At the same time, multiple TextMate syntax theme definitions are translated to CSS and included with the HTML document. Users can choose which CSS rules are applied by choosing a theme in the top left.

3.2 Presenting Code in PDF Documents through LaTeX

A different presentation of the same Rust code of fig. 3a is shown in fig. 3b embedded into this paper's text through generated LaTeX code. Just like with HTML, the Codex proof of concept can also generate LaTeX source code based on metadata in the Codex format. The two visualizations are adjusted to their medium, but both rely on the Codex metadata format.

One extra aspect of our LaTeX generator is that it can generate LaTeX for multiple files from the same project, if the metadata contains that information. Figure 4 shows source code from a different file in the same Rust project as the example in fig. 3. The two files reference each other, and when viewing this paper digitally, hyperlinks enable quick navigation between the two figures.

When there are multiple references, instead of providing a drop-down menu as shown in the generated website, in the PDF we add multiple links in superscript to parts of the code that reference multiple other locations in the code. An example of this can found in fig. 3b, fig. 4, and fig. 5.

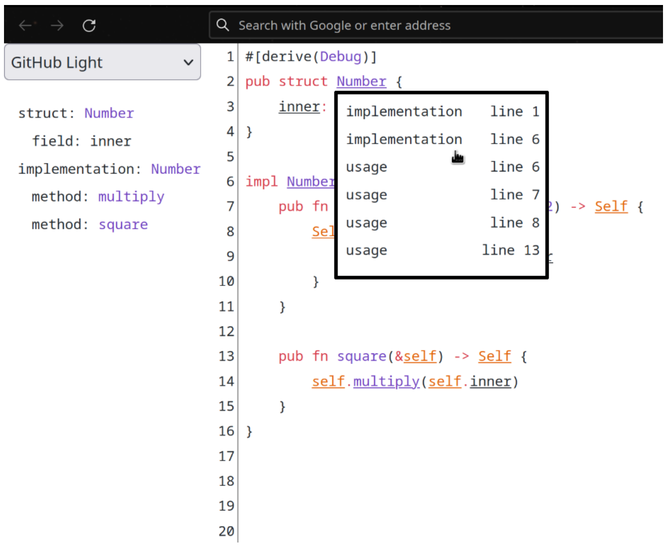
Because both the HTML presentation and this LaTeX presentation rely solely on metadata in the Codex format, both are completely language-agnostic. Although fig. 2b is mainly meant to demonstrate what the Codex metadata format looks like when it is serialized, the syntax coloring in the example itself is created using Codex. Therefore, fig. 2b is also a demonstration of how, through the Codex metadata format, we can present code written in different languages.

3.3 Generating Metadata from Existing Tools

Codex metadata can be generated either directly by instrumenting compilers, or by adapting independently developed tools. In this section, we show how we used several of such existing tools to generate metadata and how we converted this metadata into the Codex format.

TextMate Grammars. The first tool that we use to generate Codex metadata are grammar definitions. A common standard for grammar definitions supported by code editors is the one originally used by the TextMate editor [11]. Grammar definitions for many languages, both small and large,

[5]The source code for this proof of concept can be found on GitHub at https://github.com/jdonszelmann/codex



```
struct: Number
  field: inner
implementation: Number
  method: multiply
  method: square
```

```
#[derive(Debug)]
pub struct Number {
    inner:
}

impl Numbe
    pub fn                                   ) -> Self {
        Sel

        }
    }

    pub fn square(&self) -> Self {
        self.multiply(self.inner)
    }
}
```

```
implementation   line 1
implementation   line 6
usage            line 6
usage            line 7
usage            line 8
usage            line 13
```

(a) A fragment of Rust code presented in an interactive HTML webpage. The color information, reference information (underlines), outline, and warnings are derived from metadata stored in the Codex format. When a declaration in the code is referenced more than once, hovering over the underlined name opens a pop-up listing all its usages. Clicking one of the usages in the list instantly jumps the PDF viewer to the relevant code.

```
#[derive(Debug)]
pub struct Numberimpl,impl,usag {
    inner: u32,
}

impl Numberdecl,impl,usag {
    pub fn multiply(&self, other: u32) -> Self {
        Self {
            inner: self.inner * other
        }
    }

    pub fn square(&self) -> Self {
        self.multiply(self.inner)
    }
}
```

(b) The same fragment of Rust code as shown in fig. 3a, but embedded in this paper through LaTeX source code generated from the Codex metadata format. When this paper's PDF is viewed digitally, underlined elements are clickable and navigate to the referenced location. When an item is referenced more than once, superscript annotations are inserted that facilitate navigation.

Figure 3. A demonstration of our proof of concept, presenting source code on a website and in this document.

are freely available online, mostly with the purpose to be used in editors. Almost all editors either have native support for these TextMate grammar files, or have plugins which add this support.

TextMate grammars are not full context-free grammars, but instead function by applying regular expressions to fragments of source code. Based on which regular expressions matched words in the code, other regular expressions are brought in and out of scope to allow moderately complex syntaxes to be parsed. Due to their design, TextMate grammars quite often perform well even in the presence of syntax errors.

TextMate introduces a way to hierarchically classify tokens based on the token's function in a programming language. We discussed in section 2.2 how we use this function for syntax coloring and other code exploration services in the Codex format. Because we use this system for syntax coloring, translating from TextMate output to the Codex format is very simple. TextMate grammar files contain such classifiers, and we can put those directly into the Codex format.

Previous parsers using TextMate (like the one for VS Code and TextMate itself) are designed to work in combination with an editor. These proved hard to integrate in Codex, in part because they were designed to directly output color information as opposed to token classifications. For our proof of concept, we wrote a custom TextMate parser in Rust, which we have tested on grammar definitions of many large languages such as the ones for Rust, TypeScript, and Haskell.

CTAGS. CTAGS [19] is a tool that can provide primitive editor services in terminal-based editors such as Vim. To do this, CTAGS parses source files of many different languages and can generate so-called tags files from that. Vim can then read these tags files to allow programmers to search for definitions in a code base. CTAGS does not provide enough information to derive code navigation from its output. However, its output can be used to generate outlines for source files, which is what Codex does. An example of such an outline can be found in fig. 3a.

The Language Server Protocol. In section 1.1, we already mentioned LSP [14] in the context of the $m \times n$ problem. Although LSP is a protocol mainly meant for editors, we can use the information that LSP can provide and extract metadata from it. This is very different from how LSP is normally used for editor services, requiring live communication about a user's interaction with the code base.

To accomplish this, we first start a language server on the code base we analyze. Then we query this language server

```
fn print_square(n: Numberdecl,impl,usag,usag) {
    println!("{:?}", n.square());
}
```

Figure 4. This is the continuation of the example Rust code in figure fig. 3b. Code navigation enables quick navigation between the two pieces of code when reading this paper digitally.

about every token in a code base, storing all responses. We then convert the responses such that they can be stored in the Codex format, after which the language server can be stopped again. Because of the number of queries that need to be executed, this process can be rather slow, taking up a majority of the indexing time. However, for this demonstration, speed was not a priority.

Codex currently queries LSP to get both reference information and diagnostic messages. However, the LSP can provide much more than just reference information, like documentation for items, code folding points and code actions.

We have tested Codex querying an LSP, with both Rust's and Haskell's language server implementation. This is also how the reference metadata in fig. 3a, fig. 3b and fig. 2a is generated.

3.4 Producing Metadata for Small Languages: Elaine

Small programming languages, DSLs or research programming languages, often have very little tooling available. The time it costs to make tooling, such as editor services, build systems and documentation is often not worth the time. However, for educational purposes, and science communication, having some simple code exploration services like syntax highlighting and code navigation could be very advantageous.

The Codex format can help with this. The visualizations of code that Codex can generate are completely language-agnostic. That means that if the compiler or interpreter of a DSL can output metadata in the Codex format, all code exploration services in the Codex visualizations work out of the box.

To demonstrate this, we implemented a Codex generator for the Elaine language. Elaine is a domain-specific language that explores programming using higher order effects [1, 3]. Elaine is built for research, and it has a simple type checker and interpreter written in Haskell.

We modified the Elaine parser and type checker slightly, such that it output metadata in the Codex format directly from the parser and type checker. Implementing these modifications took less than three hours in total, and required less than 100 extra lines of code. The parser directly labels

```
use std;

effect Abort {
  abort() a
}

let hAbortusag,usag = fn(default) {
  handler {
    return(x) { x }
    abort() { default }
  }
};

let safe_divusag,usag = fn(x, yusag,usag) <Abort> Int {
  if eq(y, 0) {
    abort()
  } else {
    div(x, y)
  }
};

let main = add(
  handle[hAbort(0)] safe_div(3, 0),
  handle[hAbort(0)] safe_div(10, 2),
);
```

Figure 5. An example piece of Elaine code. Elaine is a research language that explores programming with higher order effects. Elaine's parser and type checker were slightly modified to output metadata in the Codex format, enabling syntax coloring and code navigation.

the syntax with similar category names as used by TextMate, without using a TextMate Grammar itself. At the same time, the type checker outputs reference information as a replacement to queries to an LSP. Since the type checker needs to resolve type references anyway, outputting the results of these resolutions is not very complicated.

For this paper, we fed the generated metadata from Elaine into Codex, which converts the code into a LaTeX representation. The result of this can be found in fig. 5.

4 Design Considerations

While designing the Codex metadata format, we encountered many interesting design choices for which we discuss several important decisions and trade-offs in this section, together with alternatives. We start with the considerations regarding how to store source code locations in section 4.1. Then, we discuss how they can be used to form spans, sections of source code, in section 4.2. This is followed with two sections, section 4.3 on how references source code within a file works, and section 4.4 on how this is extended to work between multiple files. After that, we discuss how we classify spans effectively in section 4.5. We finish with a discussion of how we can store metadata on disk in section 4.6.

4.1 Locations

In the Codex format we are describing metadata about parts of source code. Therefore, we need a way to refer to specific locations in the source files. There are two main ways in which this is commonly implemented: either as an absolute offset from the start of a file, or as pairs of ⟨line number, character offset⟩.

Line numbers more easily map the metadata to individual lines in the code editor or viewer, as they are often line-based. For an editor, this presents an advantage: line offsets only need to be adjusted whenever the user adds or removes a newline in the document. This can be beneficial for checking metadata into version control systems, or when implementing a system where metadata files can be updated incrementally (see section 5).

The downside of storing both lines and character offsets is that it may take up more storage, as every location is stored as two numbers instead of one. Additionally, to find the line that is referenced, a program would need to step through the file, counting newline characters until the target line is found. This makes it more expensive to find a particular location in a source file. For these reasons, we chose to use absolute offsets in our prototype.

Encoding. The above raises the question: in what unit do we express these absolute file offsets? One obvious choice would be to use the absolute byte offset from the start of the file. This has some downsides: an erroneous offset could point in the middle of two bytes that together form a single character, and byte offsets need to be changed whenever the code base is transformed to a different character encoding.

A character encoding specifies how the bytes in a file should be interpreted as characters by an editor or viewer. There are many character encodings. The ASCII character encoding used to be the default for a long time in the English-speaking world, where each byte represents a single character. Nowadays, the Unicode character encodings are more common, such as UTF-8 and UTF-16, which can represent more types of characters using variable-length encoding. To save space, these encodings represent common Latin characters using fewer bytes.

If offsets are expressed in bytes, and the text encoding changes, offsets will not line up anymore. For example, most programming languages use UTF-8 character encoding for their source files, but LSP, being based on JavaScript, returns offsets as a number of UTF-16 characters. This means that an extra translation step is required to make the byte offsets in the file correspond to the offsets reported by LSP.

An alternative to byte offsets, is to express offsets as a number of Unicode code points. This number is independent of the character encoding used. However, because of the variable-length nature of common Unicode encodings, finding the character at a certain offset requires iterating over the entire source code. Character lookups can be made more efficient by keeping an index mapping code points to byte offsets.

So the trade-off is that either possibly many encoding-conversions need to take place, or that character lookup requires an index to be performant. In Codex, we refer to absolute offsets by counting the number of Unicode code points from the start. This proved to be easier to keep consistent than UTF-8 byte offsets, at the cost of some performance.

4.2 Spans

To describe stretches of source code, in the Codex format we used the concept of a *span*, which denotes a selection of code. All spans have a start and end position, but there are two main ways in which this can be constructed: first, a span could contain a start and an end location, but not allow the end to be before the start. Alternatively, a span could be the combination of a start and a length, which might take up less storage space and makes it easier to verify that the length is always positive or zero. This is the approach taken for the Codex format.

Multi-part spans. One thing to keep in mind is that spans are not always contiguous in all programming languages. Agda[6], for example, can have *multi-part spans* that refer to, for example, both an opening and closing parenthesis. In that case, the metadata format should either split such spans into the component parts and store the same metadata twice, or each file offset should become a list of offsets.

4.3 References

An important aspect of the Codex metadata format is to relate metadata with spans of source code. For example, the metadata for an identifier in the source code might have a reference to the span of its associated declaration. To do that, there are numerous strategies that can be used.

To provide reference resolution in code exploration media, the metadata format needs to contain information about what locations reference which other locations. Effectively, a set of relations between parts of a code base.

Referencing locations can be done with any of the techniques discussed in section 4.1. In the Codex format, we chose to make a separate metadata entry for every single relation. However, a more efficient method might be to associate a list of reference destinations for every symbol that needs it. In that case, the start offset only needs to be given once.

In Codex, we store most relations twice. For example, one direction is from some reference to some definition, and the

[6]See https://wiki.portal.chalmers.se/agda/pmwiki.php and https://github.com/agda/agda

inverse relation would be labeled as a usage of that definition. We chose to separate these two directions, because consumers of metadata might handle the two directions differently. Additionally, it would make the format more general, as back-edges might not make sense in all situations for all programming languages.

Linking with labels. Another way to reference from one place in a piece of source code to another, would be to use a label or identifier. One piece of metadata might be on the usage of a variable, saying that that usage site is identified by the number $a123$. The location in the source code with the definition of that variable could then have some metadata with the same identifier $a123$, linking the two locations.

This technique could even allow for a complete elimination of spans. Instead, metadata could be connected to source code through lengths only, no absolute positions needed. The first metadata item in the format refers to an offset of 0 in the source code, and has a certain length. From there, every metadata item only stores a length relative to the end of the previous metadata item, assuming there are no gaps. Gaps could be labelled as lengths without metadata. This might be an efficient way to store metadata.

4.4 Referencing symbols in different files

In the Codex format, we not only want to refer to spans in the same file, but also across files.

In our prototype, we put all the metadata for a code base into a single file. We divide the file into sections, one for each file in the code base. Each section starts with the hash of that source file.

The hash makes sure that an interpreter of the metadata knows for certain that a source file and a piece of metadata really belong together. However, the hash has a second purpose. When a source file references a location in another source file, the metadata for that reference will contain the hash of the other source file. Essentially, the hash serves as an identifier for inter-file references.

Alternatively, content could be addressed by file name. This may be convenient because it makes finding the source file to which the metadata file belongs on disk easier. However, hashes are often much shorter than paths saving space, and keeping track of paths when files move around can be problematic.

4.5 Classifications

In section 2.2, we discussed a data type called `Classification`, containing a list of increasingly specific class names to identify items in metadata. TextMate [11] uses this technique to classify syntactical elements by color, but we think that using such a hierarchical system for different kinds of classifications can be useful. For example, we can label types of references using classifications. Top level categories could be `definition` and usage, and depending on the language

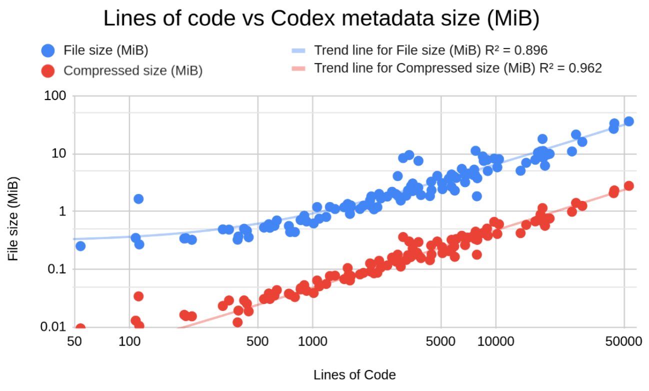


Figure 6. To give an indication for the size metadata files grow to, we used our proof of concept to generate metadata for the 100 most downloaded Rust projects. We chose Rust because for this language, our implementation currently generates the most metadata. In the plot, we compare the sizes of the projects (each in lines of code) to the file size of the serialized metadata (in MiB) on a log-log scale. In blue, the raw size is shown, while red shows the size of the metadata compressed using Zstd `v1.5.5` with its default settings (compression level 3).

a hierarchy of subcategories could be made. A simple presentation could only show definitions and usages differently, but a more elaborate presentation can support more types of references specific to only certain languages.

LSP does not do this. For example, there is the `SymbolKind` type[7]. According to LSP, there are only 26 different kinds of symbols, a list which seems to be inspired by object-oriented languages. Some languages map poorly to such a premade classification, like Haskell, which has the concept of typeclasses. Typeclasses could be classified as interfaces, but are not quite the same.

4.6 Storage

The Codex metadata format is a bridge between tools that generate metadata and tools that consume metadata. Although these two steps may sometimes follow each other directly, they do not need to. Because the metadata is static, it is possible to store the metadata and consume it later. For example, it might be desired to generate the metadata locally, commit it to version control, and have a continuous deployment job find the metadata and turn it into a static documentation website.

Although the size of metadata is not a primary concern to us, if it turns out that even for small projects we produce an unmanageable amount of metadata, our proof of concept

[7]See https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#symbolKind

might not be feasibly usable. Therefore, we ran a small experiment to give an indication of the size of the metadata we generate.

We downloaded the 100 most popular Rust projects[8], and ran our generator on each of them. These projects ranged from a few tens of lines of code to thousands. All four types of metadata relations which our proof of concept currently supports were generated.

In fig. 6, we show the results of this test, split into two categories. First, the size of the raw output, which grows at approximately a mebibyte for every 1500 lines of code. However, we also looked at how much compression could help reduce the size of the metadata.

When we use a popular compression tool, Zstd [13], the file size drastically decreases. With compression, the files only grow at about a mebibyte for every 20 000 lines of code, more than 13 times smaller. Using a generic compression tool like Zstd may be a good choice, as opposed to optimizing the format itself for storage size. While the latter option requires lots of effort to implement, complicating the data format, the former option is essentially trivial and does not depend on the data format itself.

Storing metadata for multiple files. To store metadata of multiple source code files, there exist two options. Either all metadata is combined into a single file, or the metadata is split up, possibly into a single metadata file for every source code file.

The first option is what we implemented in our proof-of-concept, since it makes finding the metadata for a source file trivial, which in turn made our implementation simpler.

However, combining all metadata also has some downsides. First, the data format becomes slightly more complicated, since we need to track what metadata belongs to what source file. Additionally, in the case where you want to check metadata into version control, diffs on the one big file can become enormous.

Splitting up metadata files has downsides as well. The biggest of which is that finding the right metadata file that contains a certain reference target can become difficult, especially if the files are potentially spread over a code base.

5 Limitations and Future Work

Our approach shows that the Codex metadata format, and its separation between information generation and presentation, is able to describe the most important aspects of a code base in a language-agnostic way. However, there are also certain aspects of Codex that still need work. For example, we do not yet support storing the metadata behind all eight code exploration services described in section 2.1.

[8]The same 100 used by the Rust playground: https://github.com/rust-lang/rust-playground/tree/main/top-crates

Next, certain aspects were out of scope for our proof of concept. Codex is not particularly fast, and it may be possible or necessary to reduce the size of metadata files.

Nonetheless, we believe that even with these limitations, Codex demonstrates its usefulness.

Apart from addressing the limitations described above, there are several questions we think should be answered in the future. The first is to see whether is it possible to encode and provide certain guarantees about the Codex data format. Right now, in our proof of concept, certain aspects of the format, like the order of metadata, are undefined. Spans of the same relation can also interleave, with one span starting before another and ending inside the other. Whether the format has these, depends on the tools that generate the metadata, while it can make implementing consumers of the format hard.

To mitigate this, we might imagine a format with certain guarantees encoded in them, and transformer tools which take metadata, and produce metadata with more guarantees. This might be by filling in missing information, sorting information based on a desired property, or splitting metadata so spans no longer interleave. Such transformer tools may make it easier to generate and to consume metadata.

Next, we think that it might be possible and even useful to make generating metadata incremental, to help with speed. A tool that generates metadata might first read in old metadata, and based on that do less work. What is interesting is that, when generating metadata becomes fast enough, it brings the capabilities of such a format very close to what LSP can do.

5.1 Other uses of the Codex format

In our proof of concept, we have demonstrated two ways to present code based on the Codex intermediate format. First in HTML, and then in LaTeX. However, we think there are many more use cases for a metadata format like the Codex format. We would like to highlight a few.

First, there is the possibility for language-agnostics documentation websites. This is, of course, related to our HTML output system. However, with a complete format with support for all the code exploration services highlighted in section 2, it might be possible to extract all the information necessary to generate websites like RustDoc for Rust or Hoogle for Haskell.

We even think that, for example, Rustdoc could directly benefit from our technique to generate HTML. In Rustdoc, it is possible to browse source code. However, the variables used in that source code do not link to their documentation items. Something that our HTML presentation already shows is possible.

Then, there is the possibility to use metadata files as a generic source to query information about a code base. Static analysis tools or linters might find such a standard form of metadata useful, making those tools instantly language-agnostic.

6 Related Work

There are multiple related attempts at solutions to the $m \times n$ problem for code exploration services. One of those we already looked at in the introduction: LSP [14]. We explained that LSP was not well-suited to more generic code exploration services, since LSP has a client-server model. However, there are more attempts worth discussing.

Narrow tools. There are numerous narrow tools that provide code exploration services. A narrow tool is a tool which either provides one code exploration service for many languages, like CTAGS [19], or one that provides many code exploration services for one language, like RustDoc. Some more examples of narrow tools are syntax coloring frameworks for websites like highlight.js and LaTeX packages like minted and listings. Although tools like these are usually very good at the narrow use case they support, they cannot support either more languages or more code exploration services. This means that many such narrow tools need to developed separately to support all services for all languages.

Stack Graphs. GitHub, a well-known code repository website, allows visitors of the site to easily navigate source code of certain programming languages (such as Python) by jumping from usages to definitions and back, even between files. They achieve this using Stack Graphs [2], a system related and derived from Scope Graphs [17, 20, 21]. However, their solution only provides code navigation, and can only support languages for which the GitHub developers wanted to direct their efforts, which will invariably not include any lesser-known languages. Additionally, language developers cannot contribute to this, even if they had the resources, as the full package of code exploration services GitHub provides are inherently tied to the GitHub website, and not publicly available.

CBS. For a language called CBS[9] a system was created to generate a documentation website based on source code [15]. At first, this might sound like another narrow tool. However, because CBS is based on the Spoofax language workbench platform [7–9], there are plans to extend the system to work for any language defined within Spoofax [16]. This does not provide a solution for popular languages that have existing tools for code exploration, but not a definition in Spoofax.

7 Conclusion

We presented Codex, a language-agnostic format for describing the metadata of a code base, with the purpose of providing code exploration services in different types of presentations, such as HTML websites and LaTeX documents. The format decouples generating the metadata from its presentation, addressing the $m \times n$ problem for code exploration services. The format allows the code to be explored at a point later in time from when the metadata is generated, even when the specific versions of tooling that was used on the code base is no longer available.

By implementing multiple language-specific generators, we showed that the Codex metadata format can be generated from several existing narrow tools (e.g., TextMate grammars, LSP, CTAGS) for various programming languages (e.g. Rust, Haskell). This includes generating metadata for a Domain-Specific Language (DSL), called Elaine, demonstrating that providing code exploration services for such languages without pre-existing tooling is feasible. The code examples in the digital version of this paper are interactive, allowing the reader to navigate between code usages and definitions. These interactive features and the syntax highlighting are generated based on the Codex metadata format, using our language-agnostic LaTeX presentation. Additionally, we demonstrated an interactive HTML presentation that can be used to explore a code base. We show that the Codex format successfully decouples languages and their metadata from their presentations.

[9]A visualization of source code generated for CBS is available at https://plancomps.github.io/CBS-beta/

References

[1] Casper Bach Poulsen and Cas Van Der Rest. 2023. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1801–1831.

[2] Douglas A. Creager and Hendrik van Antwerpen. 2023. Stack Graphs: Name Resolution at Scale. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.EVCS.2023.8

[3] Terts Diepraam. 2023. *Elaine.* Retrieved 2023-06-28 from https://github.com/tertsdiepraam/thesis

[4] Eclipse Che. 2023. *Eclipse Che™.* Retrieved 2023-06-28 from https://www.eclipse.org/che/

[5] Eclipse Foundation. 2023. *Theia.* Retrieved 2023-06-28 from https://theia-ide.org/

[6] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[7] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html

[8] Karl Trygve Kalleberg and Eelco Visser. 2007. Spoofax: An Interactive Development Environment for Program Transformation with Stratego/XT. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools and Applications (LDTA 2007) (Electronic Notes in Theoretical Computer Science)*. Elsevier. http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-018.pdf

[9] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 237–238. https://doi.org/10.1145/1869542.1869592

[10] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE portability problem and its solution in Monto. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 152–162. http://dl.acm.org/citation.cfm?id=2997368

[11] MacroMates Ltd. 2021. *TextMate for macOS.* Retrieved 2023-06-27 from https://macromates.com/

[12] Robert C. Martin. 2009. *Clean Code - a Handbook of Agile Software Craftsmanship.* Prentice Hall. http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0132350882,00.html

[13] Meta Platforms, Inc. 2023. *Zstandard.* Retrieved 2023-06-27 from https://facebook.github.io/zstd/#other-languages

[14] Microsoft Inc. 2023. *Language Server Protocol.* Retrieved 2023-06-28 from https://microsoft.github.io/language-server-protocol/

[15] Peter D. Mosses. 2019. Software meta-language engineering and CBS. *Journal of Computer Languages* 50 (2019), 39–48. https://doi.org/10.1016/j.jvlc.2018.11.003

[16] Peter D. Mosses. 2023. Using Spoofax to Support Online Code Navigation. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.EVCS.2023.21

[17] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[18] Pierre Carbonnelle. 2023. *Top IDE index.* Retrieved 2023-06-28 from https://pypl.github.io/IDE.html

[19] Universal CTAGS. 2023. *ctags.* Retrieved 2023-06-27 from https://github.com/universal-ctags/ctags

[20] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

[21] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276484