LANGUAGE AGNOSTIC CODE EXPLORATION SERVICES

JONATHAN DÖNSZELMANN

Language Agnostic Code Exploration Services

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jonathan Dönszelmann
born in Amsterdam, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`https://ewi.tudelft.nl`

Language Agnostic Code Exploration Services

Author:	Jonathan Dönszelmann
Student id:	4888766

Abstract

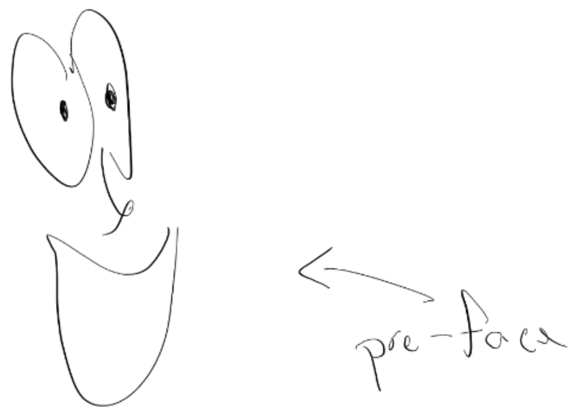
Programmers spend significantly more time trying to comprehend existing code than writing new code. They gain an understanding of the code by navigating the code base in an IDE, by reading documentation online, and by browsing code repositories on websites such as GitHub. To create rich experiences for programming languages across those various media is a large effort for developers of programing languages. This effort might be worthwhile for popular languages, but for new or experimental languages the required effort is often too large. Solutions to reduce this effort of implementing an IDE exist, such as LSP, but to reduce the effort in other places outside IDEs, we introduce the *Codex metadata format*, which separates language-specific generation of code metadata from its language-agnostic presentation. To demonstrate this approach by implementing four language-specific metadata generators (based on LSP, CTAGS, TextMate and Elaine) and two language-agnostic presentations (PDF documents and a code viewer websites) of code and metadata. To demonstrate different kinds of code metadata, we implemented four code exploration services: syntax colouring, code navigation, structure outline, and diagnostic messages. We show that with the Codex metadata format, we can decouple the metadata generators from the presentations.

Thesis Committee:

Chair:	Prof. dr. Koen Langendoen	Embedded Systems
Committee Member:	Dr. Jesper Cockx	Programming Languages
University Supervisor:	Ir. Daniël A. A. Pelsmaeker	Programming Languages
University Supervisor:	Ir. Danny M. Groenewegen	Programming Languages

Preface

They say a picture is worth a million words, and yet it felt appropriate to add a few words to the original preface Terts Diepraam jokingly made. Because that is not the only artistic contribution I have to thank him for. Every time I sent him this document for feedback, I got back a front page with even more drawings than the time before, together with a reviewed thesis for which I am incredibly grateful. Similarly, I am grateful for the help and support of my (other) friends, family and supervisors. In particular Laura, V, Ricardo, Thijmen, Anne and George, also for all the games we have played and dinners we have had, without which this work would not have been possible.



Original pre-face by Terts Diepraam

You may be reading this document on paper. Generally, I also prefer this. However, be aware that some parts of this thesis are interactive when read digitally in a PDF reader, so for a full experience I do recommend that.

Finally, this is a programming languages thesis after all. The rest of this document does not mention the word monad, but it did not feel complete without using the word at least once so there we go.

Jonathan Dönszelmann
Delft, the Netherlands
October 16, 2023

Contents

Introduction

As programmers, we spend much more time reading code than writing it. We try to get acquainted with a 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 (Martin 2009).

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 internalise code from offline paper-based publications.

In this thesis, we refer to this as *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*. These form a subset of the more well-known *editor services* (Erdweg, van der Storm, Völte, et al. 2013) which only support a user in editing code, but exclude services that are only useful when writing code.

Code exploration services range from simple yet effective syntax colouring, 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 *code exploration media*: places where code might be explored, such as on documentation websites and in books. The code exploration services and their presentations need to be adjusted to meet the limitations of the specific medium. For example, syntax colouring is feasible in a printed book, but one cannot instantly jump to a definition.

There are existing solutions that facilitate code exploration. Developers of most major programming languages have spent effort to implement their own editor plugins that provide syntax colouring and additional editor services for various editors. 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 Haddock, and Agda can output code as rich HTML and even LaTeX code. However, all these solutions are very narrowly applicable to only those 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 time and effort 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, there is also CTAGS (CTAGS 2023), which is a tool that generates an index of all identifiers found in a code base and which

[1]Explore the Linux Kernel source code at: `https://elixir.bootlin.com`.

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 is reminiscent of a problem known as the *IDE portability problem*, where for m languages to provide services for n editors require $m \times n$ implementations and related effort (Keidel, Pfeiffer, and Erdweg 2016). One solution to the IDE portability problem is a system that *decouples* editors from language-specific providers of editor services, which is what a system like Language Server Protocol (LSP) does. All languages that expose a *language server* implementation can provide editor services to all editors with an LSP client. With LSP, m languages and n editors need to support LSP, transforming the effort required from $m \times n$ into $m + n$.

The IDE portability problem also presents itself in other media. There are maybe a dozen major code editors, but a nearly limitless number of places, such as many different websites, where having better code exploration services makes sense. The $m \times n$ problem is thus even bigger for code exploration services than for editor services! In this thesis, we will therefore aim to answer the question:

How can we provide code exploration services outside editors in a way that decouples programming languages from code exploration media?

Codex

To answer that question, 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 any 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 not available. We discuss the reasoning that led us to the Codex format, describe what choices we made in the progress, demonstrate our implementation of four prototype *generators* of the Codex format and two different rich prototype *presentations* of the Codex format and finally reflect on our experience making these prototypes. The source code for Codex can be found online at https://github.com/jdonszelmann/codex.

Contributions In summary, our contributions are:

- a thorough introduction of editor services and the IDE portability problem than was given in the paper;
- rigorous definitions of the terms *code exploration* and *code exploration services*, which we introduced;
- a detailed study of existing tools that provide code exploration services and how they relate to the Codex metadata format;
- the Codex metadata format, a tractable solution to the $m \times n$ problem for code exploration services, able to describe 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.

These contributions align with the contributions of a paper which we submitted to Software Language Engineering (SLE) 2023 called “Codex: a Metadata Format for Rich Code Exploration” (Dönszelmann, Pelsmaeker, and Groenewegen 2023) (See appendix A). Unfortunately, this paper was not accepted, the reason was, in part, how we treated related work. This thesis also serves as an extension of that paper, containing much more detailed related work and reasoning behind decisions.

The rest of the thesis is structured as follows: We start in chapter 1 with an overview of what editor services are, together with a detailed explanation of the IDE portability problem. This leads into chapter 2, where we define the terms *code exploration*, and *code exploration services* We relate these terms to what we learned about editor services in the previous chapter, go over existing tools for code exploration services and establish that a similar problem to the IDE portability problem exists in this context. With this theory behind us, we will design our solution in chapter 3, and demonstrate this solution called Codex in chapter 4 We will discuss the observations we made while making this solution, together with the tradeoffs we made in chapter 5. Finally, we present future work in chapter 6 and conclude this thesis in chapter 6.

Chapter 1

Editor Services

The main subject of this thesis is what we call *code exploration services*. However, we think that we cannot discuss code exploration services without first talking about the concept that forms the foundation of that idea, which is the focus of this chapter. We will first define what editors and editor services are, after which we go over the effort required to implement editor services. We finish with the IDE portability problem, which is a problem central to the rest of this thesis.

Although programmers used to manually punch their programs into paper punchcards, or before that, even changed the wiring of a computer just to program it, this has not been common practice in the last 50 years. Instead, programmers write computer programs *for* a computer, *on* a computer, using a computer program commonly called an *editor*. Using editors for programming makes the process much easier, and vastly more accessible to everyone who has a computer.

Early on, there was *ed* (Thompson 1969), short for 'editor'. Ed was one of the standard tools included on all unix operating systems, and can still be used on many modern computers. However, ed was not very user-friendly, and is infamous for its error codes. For all kinds of different errors, ed simply prints a `?` without any further context (IEEE/The Open Group 2017).

A lot has changed since then, and editors have become significantly more advanced. First with editors like *vi*, *vim* and *emacs*, which were an improvement in user experience, if only for the fact that these allow their users to see the document while they are editing it, a feature *ed* notably lacked. Still later, Graphical User Interface (GUI) editors entered the scene, like *Notepad*, *Notepad++* and *Sublime Text*, and many more.

There are also more sophisticated editors that help programmers while they write their programs, such as *IntelliJ*, *Eclipse* and *Visual Studio Code*. These often get called Integrated Development Environments (IDEs), referring to the fact that they allow users to do much more than just edit source code[1]. What makes an IDE more sophisticated, is that an IDE often knows much more about the language that is being edited in them and can help programmers with writing their program. For example, an IDE may know about the different libraries you use or might want to use, and can refer to those libraries' documentation while the programmer uses items from them in your program.

To talk about the capabilities of an IDE, we can talk about what is called an *editor service*, a term first used in literature by Kats, Kalleberg, and Visser (2008). We can say that a 'smart'

[1]Of course, some simple editors like vim and emacs can be extended near endlessly by users to operate more like an IDE. Many people, in fact, swear by these editors, considering any added features nothing more than a distraction.

editor such as an IDE provides many *editor services* to help programmers write their programs. However, as we will see, some things we classify as an 'editor service' can also be found in many editors often considered 'simple' editors, not just in IDEs, making the distinction between an IDE and simpler editors a bit fuzzy[2].

1.1 An Overview of Editor Services

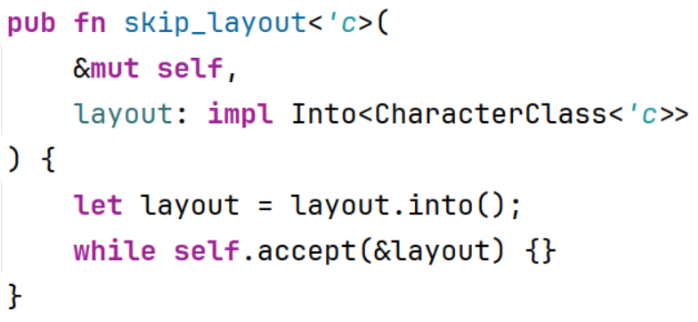
Editor services, the tools which editor provide, vary widely in kind and capability between different editors. Nevertheless, attempts have been made to make a comprehensive list based on the different kinds of services provided by many editors and IDEs. One such list is the one by Erdweg, van der Storm, Völte, et al. (2013) in "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge". Here, a feature model for a *language workbench* is given, which we will discuss further in section 1.3.2.

Important is the clear relation between language workbenches and editor services. In fact, Erdweg et al.'s feature model considers it one of six main features of what a language workbench is. Therefore, as part of defining what a language workbench is, they also list categories of editor services.

The feature model by Erdweg et al. was later adapted by Pelsmaeker (2018) into a list of just editor services. Their derived list further divides the services into larger categories, and also renames a few to be more intuitive. For example, syntax highlighting was renamed to syntax colouring as highlighting would imply a change in background colour. Finally, Pelsmaeker also provides a useful overview of many popular editors and lists which editor services each supports.

Based on these two sources, we compiled the following list of editor services, together with a short explanation of what each means.

Syntax Colouring Assigns parts of a program colours based on their syntactical meaning. This depends on the language that is being written in, which can sometimes change within a single document. For example, inline SQL database queries have a different syntax from the programming language surrounding the query. A related concept is semantic colouring[3], where colours are assigned based on semantic meaning such as by giving all uses of a variable the same colour. To provide this service for a language, the editor needs to know the syntax of that language and a colour scheme that assigns certain syntactical elements those colours.



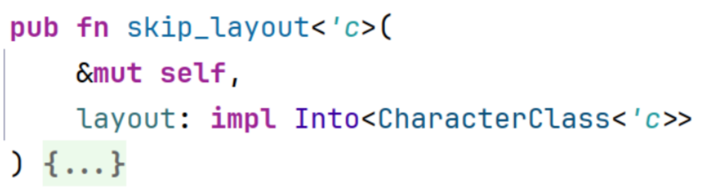
```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>
) {
    let layout = layout.into();
    while self.accept(&layout) {}
}
```

A function with coloured syntax.

[2]This is made worse because IDE is sometimes used as a marketing term.

[3]Which is more commonly called semantic highlighting, but we think it should be called Semantic Colouring just like Syntax Colouring.

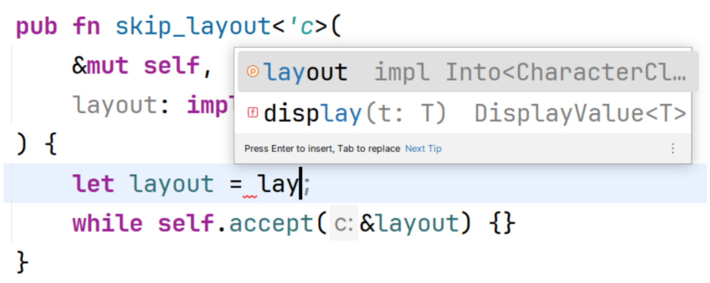
Code Folding Hides parts of a program to reduce visual clutter. Often this is presented to users as collapsable blocks of code, based on the syntax of that block, for example, the body of a function or the contents of a conditional expression. To provide this service for a language, an editor needs to know about certain points in the syntax of a language where logical blocks start, and what constitutes a logical block may be different for different languages.



```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>
) {...}
```

The function body of `skip_layout` folded away.

Code Completion Suggests code fragments as a programmer is writing their program. Simple code completion can be based on occurrences of words in the same document or project, a technique often employed by simpler editors. However, such simple completion often finds nonsensical completions. More sophisticated code completion can include knowledge about a language's scoping and importing rules, suggesting only completions that are valid based on context. Recently, AI-based code completion has also established itself, with programs such as GitHub Copilot (Github 2023a) and other large language models. These tools provide completion based on a training set containing code written by others, which has sparked some debates related to whether some advanced code completion could at some point be considered plagiarism. In fact, lawsuits (*GitHub Copilot litigation* 2022) have been filed over this, though no final verdict has been given at the time of writing. To provide this service for a language, an editor needs to know about all items in scope at the location where the programmer writes, and be able to select likely candidates based on various factors. These factors may involve the type of items around where the suggestion is made, what values are syntactically valid and what the programmer has already typed, which may be a prefix of what they want the completion to be.

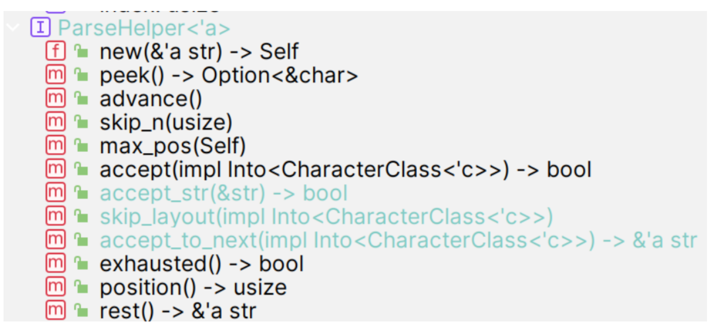


```
pub fn skip_layout<'c>(
    &mut self,
    layout: imp
) {
    let layout = lay;
    while self.accept(c: &layout) {}
}
```

```
layout   impl Into<CharacterCl…
display(t: T)   DisplayValue<T>
Press Enter to insert, Tab to replace  Next Tip
```

Completing `layout` while writing `skip_layout`.

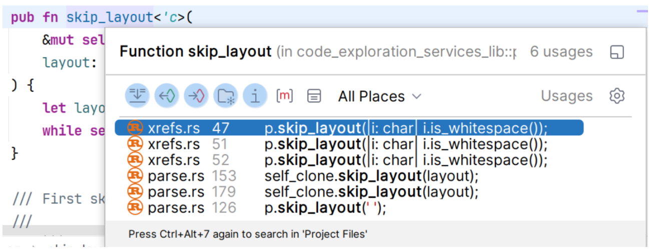
Outline An overview of the high-level structure of a program, like an index. Such an overview can often display what functions, classes and interfaces are defined in a file, and allows for quick navigation to their definition sites. To provide this service for a language, an editor needs a simple model of the syntax of a language. This model must have enough context to work out what items are present and are important enough to be included in the outline, and whether one such important item is nested inside another.



```
ParseHelper<'a>
    new(&'a str) -> Self
    peek() -> Option<&char>
    advance()
    skip_n(usize)
    max_pos(Self)
    accept(impl Into<CharacterClass<'c>>) -> bool
    accept_str(&str) -> bool
    skip_layout(impl Into<CharacterClass<'c>>)
    accept_to_next(impl Into<CharacterClass<'c>>) -> &'a str
    exhausted() -> bool
    position() -> usize
    rest() -> &'a str
```

The outline of the file with of `skip_layout`.

Code Navigation Provides quick navigation between related parts of a program. For example, between definition sites and usages of elements in a code base. To provide this service for a language, an editor needs a list of related locations in the program for every identifier in the program. These relations between locations, in essence, form the edges in a graph where nodes are source locations. Edges in this graph could also contain information about what kind of relation is represented. For example, different kinds of edges may relate a variable to other usages of that variable, or to the initial definition of a variable.



```
pub fn skip_layout<'c>(
    &mut sel
    layout:
) {
    let layo
    while se
}

/// First sk
///
```

Function skip_layout (in code_exploration_services_lib::ɾ 6 usages

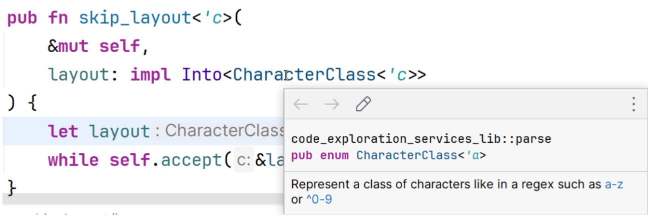
All Places Usages

```
xrefs.rs  47   p.skip_layout(|i: char| i.is_whitespace());
xrefs.rs  51   p.skip_layout(|i: char| i.is_whitespace());
xrefs.rs  52   p.skip_layout(|i: char| i.is_whitespace());
parse.rs  153  self_clone.skip_layout(layout);
parse.rs  179  self_clone.skip_layout(layout);
parse.rs  126  p.skip_layout(' ');
```

Press Ctrl+Alt+7 again to search in 'Project Files'

A list of links to all usages of `skip_layout`.

Documentation A broad range of services that provide access to the documentation associated with code. Examples include: showing documentation of items when interacting with item names such as hovering over them, applying markup to documentation text embedded in code and linking to online documentation. To provide this service for a language, an editor needs the documentation for all items in a program which have documentation available. This documentation does not necessarily need to be separately stored. If documentation is part of source code itself, the editor could dynamically load relevant documentation by following references to definition sites which can be found through code navigation.



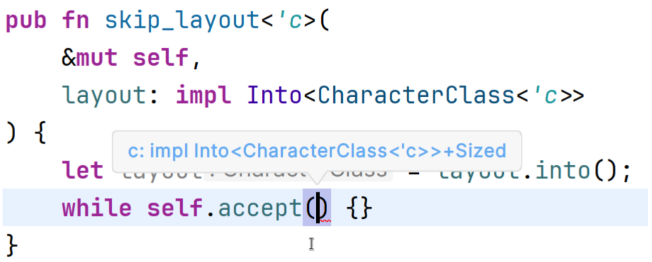
```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>
) {
    let layout : CharacterClass
    while self.accept( c: &la
}
```

```
code_exploration_services_lib::parse
pub enum CharacterClass<'a>
```

Represent a class of characters like in a regex such as a-z or ^0-9

The documentation of the `CharacterClass` type while hovering over it.

Signature Help Especially in strongly typed languages; shows information about an element's definition while a programmer is typing a usage of that element. For example, when a programmer begins writing a function call, signature helps shows the expected parameter types and names for that function. This service can be related to documentation in more weakly typed languages, where types are sometimes part of comments or annotations that are essentially equivalent to comments. Some editors, such as the JetBrains IDEs, might even go a step further, and provide interactively fillable fields.

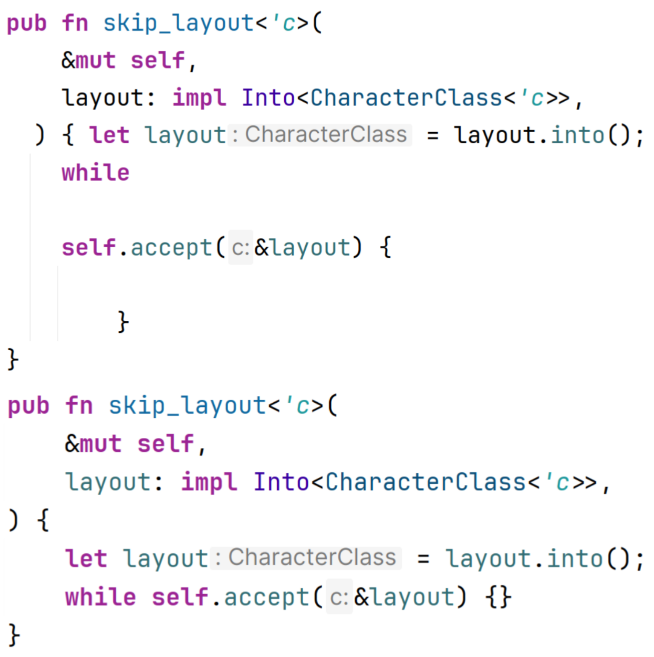


```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>
) {
    let                                    .into();
    while self.accept() {}
}
```

c: impl Into<CharacterClass<'c>>+Sized

A popup with information on how to give `accept` parameters.

Automatic Formatting A refactoring that rewrites source code to adhere to a certain style guide. Some languages define a standardised style guide like Python, which has the PEP8 (2001) standard, or have standardised tooling to format code, like Rust, which has the Rustfmt (The Rust Programming Language 2023b) project. Otherwise, many editors support reading special EditorConfig files, which contains style rules for the code in a code base. To provide this service for a language, an editor needs to know about such style guides for languages, and must then be able to apply those rules to change unformatted code into formatted code with precisely the same meaning.

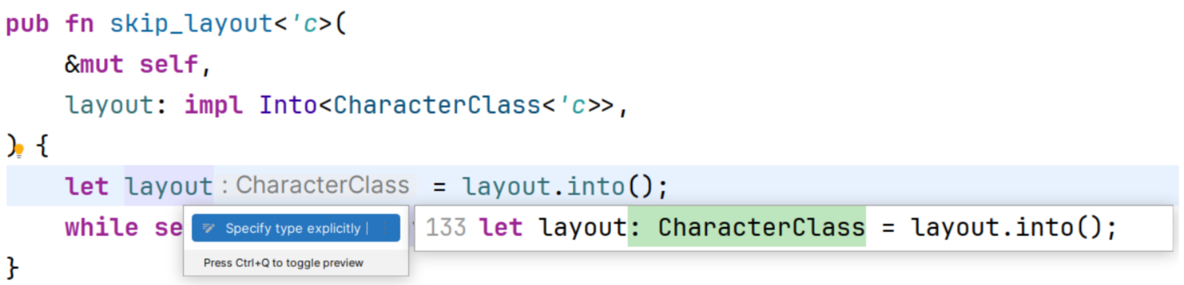


```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>,
  ) { let layout: CharacterClass = layout.into();
    while

    self.accept(c: &layout) {

        }
}
```

```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>,
) {
    let layout: CharacterClass = layout.into();
    while self.accept(c: &layout) {}
}
```

`skip_layout` before and after automatic formatting.

Code actions A broad range of language-specific actions and refactorings that can be applied to programs. What code actions are relevant may change per language. For example, a language that has syntactical macros might allow users to expand macro usages through a code action. As such, to provide code actions for a language, an editor needs the ability to execute language-specific actions and transformations on user-selected code.

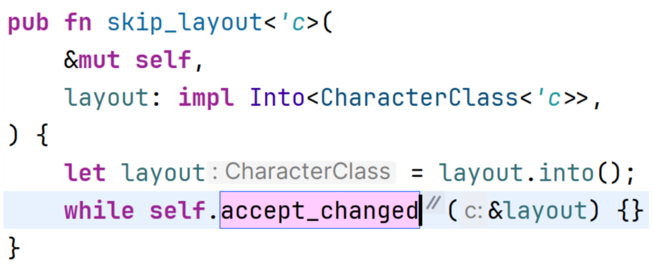


```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>,
) {
    let layout: CharacterClass = layout.into();
    while se
}
```

```
let layout: CharacterClass = layout.into();
```

A code action is being selected that will insert the `CharacterClass` type explicitly, instead of the type being infered.

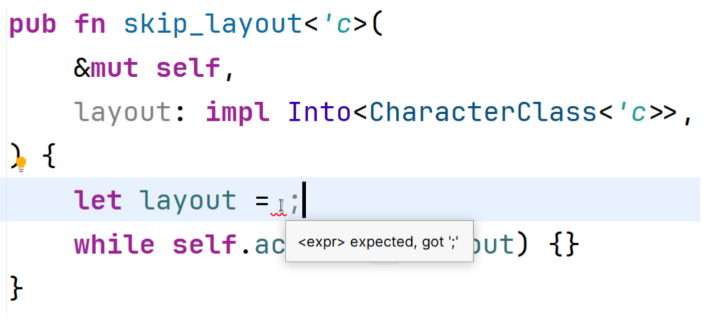
Rename Refactoring A refactoring that performs automatic renaming based on context and scoping rules of language. This is sometimes called structural search and replace (JetBrains 2022). This implementation of such a service is closely related to code navigation; to provide this service for a language, an editor needs the ability to rename a single identifier and all related identifiers from the list of related items.



```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>,
) {
    let layout: CharacterClass = layout.into();
    while self.accept_changed(c: &layout) {}
}
```

The `accept` method, in the process of being renamed from a usage. After the renaming is confirmed, the definition and all usages of will change as well.

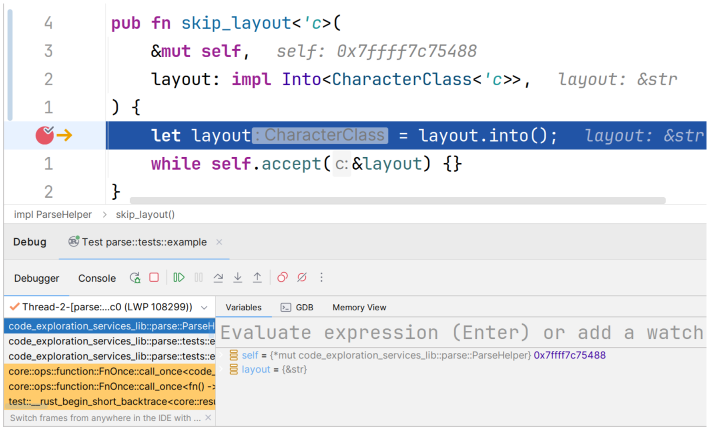
Diagnostic Messages Gives real-time feedback, maybe from a compiler or linter, often as an overlay on source code. To provide this service for a language, an editor needs to interpret the output of external static analysis tools (which could be a compiler).



```
pub fn skip_layout<'c>(
    &mut self,
    layout: impl Into<CharacterClass<'c>>,
) {
    let layout = ;
    while self.ac <expr> expected, got ';' out) {}
}
```

skip_layout at a breakpoint in the middle of the function, with values of variables show inline.

Integrated Debugger Allows a programmer to debug a program in an editor by showing the runtime state of that program together with the source text in the editor at a particular point in its execution. For example, a debugger can display the location in the source code that is currently being executed and overlaying the value of variables. To provide this service for a language, in theory, an editor needs debug symbols associated with a program that is being debugged as well as information on how to display runtime values. However, in practice an editor might simply interface with a dedicated debugger program like *gdb* and provide a graphical user interface to interact with the debugger to programmers.

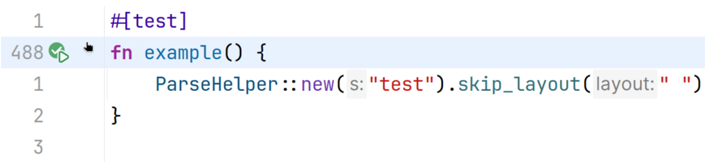


```
pub fn skip_layout<'c>(
    &mut self,   self: 0x7ffff7c75488
    layout: impl Into<CharacterClass<'c>>,   layout: &str
) {
    let layout: CharacterClass = layout.into();   layout: &str
    while self.accept(c: &layout) {}
}
```

```
impl ParseHelper > skip_layout()
Debug   Test parse::tests::example
Debugger   Console
Thread-2-[parse:...c0 (LWP 108299))
code_exploration_services_lib::parse::ParseH
code_exploration_services_lib::parse::tests::e
code_exploration_services_lib::parse::tests::e
core::ops::function::FnOnce::call_once<code_
core::ops::function::FnOnce::call_once<fn() ->
test::__rust_begin_short_backtrace<core::resu
Switch frames from anywhere in the IDE with ...
Variables   GDB   Memory View
Evaluate expression (Enter) or add a watch
self = {*mut code_exploration_services_lib::parse::ParseHelper} 0x7ffff7c75488
layout = {&str}
```

A test of skip_layout with a little green arrow in the sidebar to run just that test. The green arrow has a checkmark to show this test passed in the last run.

Integrated Testing Shows the status of tests, and allows users to run tests by selecting them in the editor. To provide this service for a language, an editor needs to interpret the output of specific testing tools for that language. Sometimes such tools are standardised like in Rust, for example, which has a built-in test runner. However, in C there is no such standard test runner, and the editor may need to support multiple different testing tools for a single language.



```
#[test]
fn example() {
    ParseHelper::new(s: "test").skip_layout(layout: " ")
}
```

Before the expression is filled in, a syntax error is shown.

1.2 Implementing Editor Services

To provide editor services, an editor needs information about the language it is providing these services for. Exactly what kind of information is required depends on the service. For example, for an editor to support syntax highlighting for a language, the editor needs knowledge about the language's syntax, which might be derived from a grammar specification.

A service that only needs syntactical knowledge about a language might be considered relatively simple though. Providing diagnostic messages, code navigation and code completion interactively might require constantly re-running analyses on source code while the user is typing, which can be enormously costly. Although such analyses should give outputs that closely resemble that of what the compiler or interpreter would show, a regular compiler is most likely not the right tool for this job[4].

A regular compiler, also called a *batch compiler*, is good at turning large batches of source code into machine code quickly (Katzan Jr. 1969; Kladov 2022). These batches are, for example, the files of an entire code base and its dependencies. In contrast, the kind of program that analyses source code for an editor has different goals. In editors, users mostly care about getting as much feedback as possible, as quickly as possible after making small changes to a program.

We will call such a compiler an *editor compiler*, although that is not an established term[5]. A user expects an editor compiler to give live feedback while they type, with latencies of at most a few seconds. Often, the only way to achieve these latency goals is by being extremely incremental, caching as much information from previous analyses to provide real-time feedback.

Furthermore, an editor compiler is often expected to be very resilient to errors. As a user writes a program, they often leave the program in a syntactically invalid state halfway through typing a line. Still, services such as code completion are expected to provide reasonable suggestions for these half-written programs.

And yet, an editor compiler still has much of the complexity a regular compiler has to deal with. It still has to parse and type check a language in a manner that is often expected to be consistent with what the regular compiler for that language would produce. Therefore, implementing such tools can be a complex task.

1.3 The IDE Portability Problem

Because editors need knowledge about a language to provide editor services for that language, not even the most advanced editor can support all languages. Lesser used languages and Domain-Specific Languages (DSLs) often have poor support from editors out of the box. For that reason, many editors provide a way to be extended by users, often in the form of a plugin system. That way, an author of a new language can also create a plugin providing support for that language in a specific editor, which means the editor authors do not need to spend time supporting all languages.

However, shifting the responsibility like this creates a new problem. Now a language author, or the community around a language, becomes responsible for providing plugins for their languages in all kinds of editors, which often work subtly differently. Again, this means that smaller languages and DSLs will be inherently less supported as they may not have the resources to implement and maintain specific tooling for multiple editors. As discussed in the introduction, this is a problem of responsibility, where either a language has to support n editors or an editor has to support m languages, is called the IDE portability problem by Keidel, Pfeiffer, and Erdweg (2016). However, sometimes the problem is simply referred to as the $m \times n$ problem, framing the problem as a kind of complexity. There have to be $m \times n$ implementations providing editor services for m languages and n editors.

[4]An interpreted language might not have a *compiler*. However, an interpreter performs many of the steps relevant for this argument. Source code still needs to be parsed, and typechecked (even though that may happen at runtime for some languages).

[5]It is debatable if compiler is the right word for this kind of tool. I argue a compiler is fitting name, since an editor compiler has to solve many similar problems to batch compilers even though in the end no machine code is compiled. Problems such as name resolution and type checking still need to happen as normal.

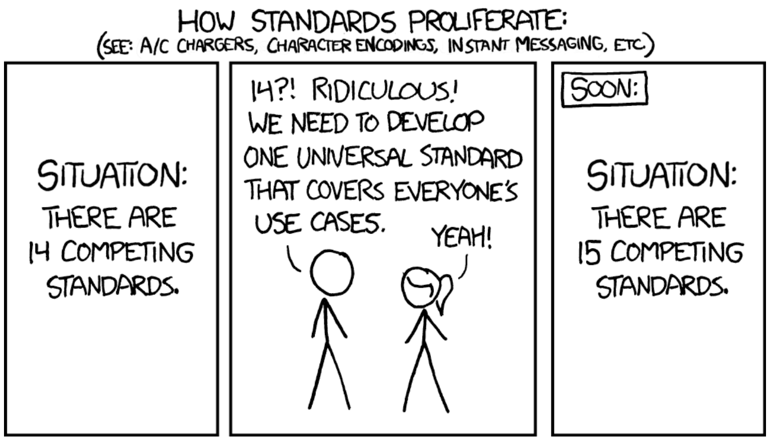

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 1.14: The XKCD 'Standards' comic, from `https://xkcd.com/927/`

In "Portable Editor Services", Pelsmaeker (2018) investigates this issue, and proposes a possible solution: Adaptable Editor Services Interface (AESI). AESI is a system to create editor plugins, motivated by the desire to separate concerns between the Spoofax language workbench, and the editors supported by Spoofax. They introduce a kind of common interface through which languages and editors can talk. If all editors only need to support one interface to talk to all languages, and if all languages only need to support one interface to talk to all editors, the $m \times n$ problem is essentially transformed into an $m + n$ problem.

Although the idea of a single common standard might be good in principle, it only works when all parties agree on the same standard. The well known XKCD comic in figure 1.14 illustrates this problem. And yet, there does seem to be one standard for editor services that is becoming reasonably universal: LSP, which we will discuss in the next section.

1.3.1 Language Servers

The idea of solving the $m \times n$ problem by providing a single standardised interface through which languages and editors can talk is also the basis for LSP, a protocol developed by Microsoft (2022a). Although AESI lacked traction, LSP is doing rather well in this regard, with support in many editors and languages, like for example NeoVim and Fleet, and many more (Sourcegraph 2023). Importantly, Microsoft's own Visual Studio Code, one of the most popular editors, natively supports LSP which helped the protocol gain popularity (Carbonnelle 2023).

LSP works on a client-server model. On one side there is the *language server*, which is a program that provides language-specific implementations for kinds of editor services. The list of supported services is almost equivalent to our list given in section 1.1. A language server usually keeps running in the background for as long as the LSP client, (i.e. the editor) is open. This allows the language server to keep state between requests.

The LSP client and server are two different processes. They communicate by exchanging standardised JSON messages using JSON-RPC. An editor that acts as an LSP client can dynamically make requests to the server based on the user's actions in the editor. For example, the editor might tell the server that a file has been opened, to which the server might respond that it is analysing that file now. These requests are asynchronous, meaning that multiple requests can be in progress at the same time, and it is even possible for the language server to make a request back to the editor[6]. The asynchronicity also makes sure that the editor does not have to block while waiting for requests to finish.

[6]This can happen when the language server 'creates' a progress bar, for example for how long it will take to index a project. The language server them makes a request to the editor to tell it it will start sending progress notifications. The editor must then respond if it wants to receive these.

Although LSP defines a communication protocol, it does not define the channel over which messages are sent. One common approach is that the editor spawns a language server as a subprocess, communicating with it over the standard *in* and *out* streams. However, many other communication channels can be used, including networked channels like websockets. For example, Eclipse Che and Theia are web editors, which make requests to a language server over the internet (Eclipse Foundation 2023a; Eclipse Foundation 2023b). This does have the disadvantage of some added latency caused by the network.

1.3.2 Language Workbenches

Erdweg, van der Storm, Völter, et al. (2015) define that "language workbenches are environments for simplifying the creation and use of computer languages." In a language workbench such as Spoofax, Xtext and Rascal, users can write down a description of a language using several DSLs (Fowler 2005; Kalleberg and Visser 2007; Kats and Visser 2010; Eysholdt and Behrens 2010; van der Storm 2011). Such a specification may consist of syntax rules of a language, a description of how the type system of the language is supposed to work, and rules on how to generate machine code. Then, based on this specification, the language workbench can derive an entire compiler and editor, complete with editor services.

With a language workbench, the workbench itself essentially functions as a common interface between editors and languages. If a language workbench supports deriving editor service implementations for n editors, then the workbench can use its derivation rules to generate these implementations for m languages specified in the language workbench, providing one solution to the $m \times n$ problem.

Of course, only languages that have a specification in the language workbench can benefit from these automatically derived editor services. This effectively excludes most big languages, which often have a custom-made compiler. Luckily, those bigger languages often have the resources to implement a language server or provide editor services for several editors. However, the story is different for smaller DSLs. For those, using a language workbench could be valuable, exactly because the workbench ensures that the language immediately has editor support.

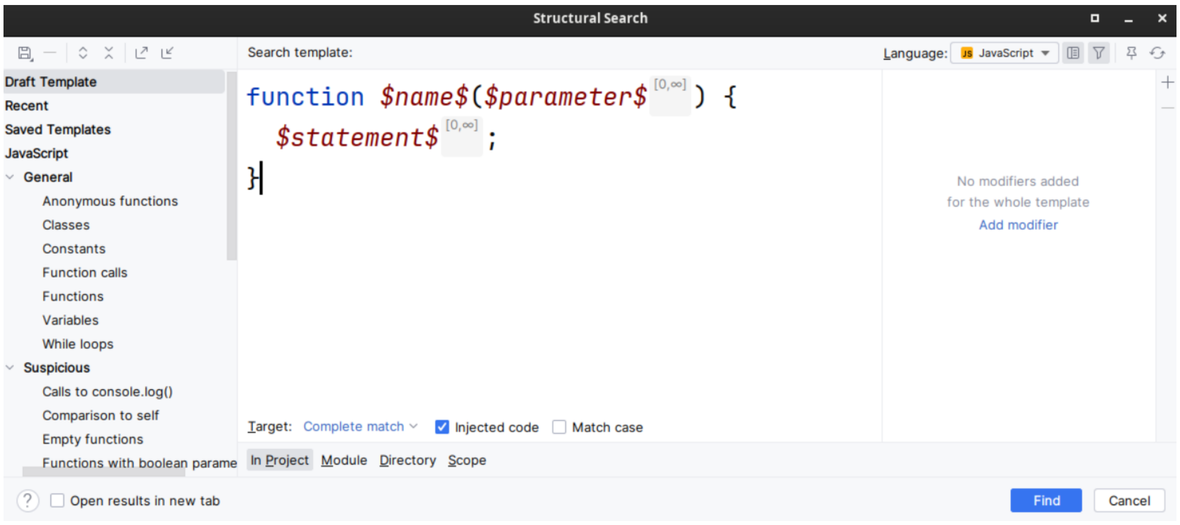
While the Spoofax language workbench does not support many editors to generate editor services for[7], Xtext, a different language workbench can generate entire Language Server implementations supporting syntax colouring, code folding, code completion, an outline, code navigation, signature helpand code actions based on a description of a language. That means that a language made in Xtext automatically has editor support in all editors that work with LSP.

1.4 Code Search

Although our list of code exploration services has its roots in a list of editor services, which itself has its roots in frequently cited theory by Erdweg, van der Storm, Völte, et al. (2013), we think that one feature may be missing from the original list. That feature is *search*. That search is missing is interesting, because the feature is pretty much universally supported by editors, even 'dumb' ones. Even many websites, where we showed that often relatively few services are supported, offer advanced search options.

We suspect that the reason that search is missing from the work by Erdweg, van der Storm, Völte, et al. (2013) is the following: searching through a code base is often language agnostic. Therefore, an editor does not need any language specific knowledge to provide it, and can also provide it for documents not written in any programming language. As

[7]Only Eclipse is properly supported, although attempts have been made to also support LSP-based editors and IntelliJ (Pelsmaeker 2018))



```
function $name$($parameter$[0,∞]) {
  $statement$[0,∞];
}
```

Figure 1.15: Structural search in IntelliJ: an example of a query for any JavaScript function with any number of parameters. Other options are supported as well, like searching for specific parameter or return types.

a consequence, there is no IDE portability problem with search. However, in our opinion we should consider it to be an editor service: it is a service an editor provides, that helps programmers edit programs.

Furthermore, the claim that search is language-agnostic is not always true. Intellij, and other JetBrains IDEs support a service which they call 'structural search'. We show an image of what this looks like in figure 1.15.

Structural search is only supported for a limited number of languages, because it is not language agnostic. Through the menu, a programmer can construct a kind of pattern with types and parameters, which matches elements of a programming language for syntax-aware search.

In the rest of this thesis, we will assume search to be an editor service.

Chapter 2

Code Exploration Services

In this chapter, we introduce and define the terms *code exploration* and *code exploration services*. We explain the reasoning that leads us from editor services towards code exploration services, discuss the status quo of code exploration services and discuss the possibility of more, different code exploration services. Finally, we establish that a problem similar to the IDE portability problem exists with code exploration services.

2.1 Code Exploration

As the name *programmer* implies, a programmer's job is to program: writing programs. Programs are written in editors, which provide certain editor services. The goal of editor services, as discussed in chapter 1, is to simplify writing programs by helping the programmer in various ways.

However, programming is not just about writing programs, but also about reading and understanding programs. This can be a time-consuming process, especially when reading code written by other people. In fact, it has been estimated that programmers spend ten times as much time reading code than writing it (Martin 2009).

Moreover, reading code is more complex than for example reading an article or a book. A book has a decidedly linear structure; you can start on the first page, and slowly work your way to the last. Although small sections of programs might be linear, an entire program often is not, unless very carefully crafted. Function calls, imports and external resources such as online documentation all mean that reading code requires frequent shifts in focus between sources of information. Therefore, a more descriptive name for interpreting and understanding a program might be *exploring*, not *reading*, and from now on we will refer to it as such.

Definition 1 ***Code Exploration*** *is the process of interpreting programs, with the goal of understanding how it works.*

Interestingly, the fact that reading code is different to reading natural language has also been observed in Artificial Intelligence (AI). In AI research there is something called the *Naturalness Hypothesis* (Allamanis et al. 2018), which says that programs and natural languages are not so different. Therefore, it is said, natural language models (which, for example, can summarise written texts) should also work well on programs. This hypothesis seems to hold well on a small scale, but Ben-Nun, Jakobovits, and Hoefler (2018) claim that on a large scale it does not hold. Because natural language models often work linearly, processing tokens sequentially, they are not well adapted to switching focus as required to interpret code. In their paper, Ben-Nun, Jakobovits, and Hoefler present their own approach, which can deal with this problem better.

2.1.1 Passive and Active Code Exploration

This definition of code exploration we gave above is extremely broad. Many things can be considered code exploration, not just reading functions from top to bottom. For example, attaching a debugger to a running program, and observing the program's behavior can be considered code exploration. Similarly improving one's understanding of a program could be better understood by simply running it and testing different inputs.

The latter form of code exploration is more interactive than simply browsing a program's source code or associated documentation. It requires a programmer to interact with the program at runtime to learn more about it. For the rest of this thesis, this difference will become important. Therefore, to make the distinction between these forms of code exploration clearer, we will use the words 'active' and 'passive' to describe code exploration according to the two definitions below.

Definition 2 ***Active Code Exploration*** *is code exploration that involves interacting with the program as it executes.*

Definition 3 ***Passive Code Exploration*** *is all non-active code exploration. It does not involve an interaction with an executing program.*

2.2 Code Exploration Services

In chapter 1, we looked at how editor services can help programmers to create code. However, many editor services do not specifically help with *writing* code. Instead, they help programmers *explore* code, making it easier to understand a code base, which could then help a programmer write code. Code navigation is maybe the most obvious example of such an editor service, that mainly helps with code exploration and not writing code. We will call these kinds of editor services *code exploration services*.

Definition 4 ***Code Exploration Services*** *are a subset of editor services that help with code exploration.*

Documentation, outline and syntax colouring are also examples of editor services which help mainly with code exploration. Notice though, that according to the definitions above, these examples aid with *passive* code exploration instead of *active* code exploration. On the other hand, integrated debugging and testing, are editor services which help with active code exploration instead. Just like we split up the definition of code exploration, we can also split up the definition of code exploration services as follows:

Definition 5 ***Active Code Exploration Services*** *are code exploration services that help with active code exploration.*

Definition 6 ***Passive Code Exploration Services*** *are code exploration services that help with passive code exploration.*

Figure 2.1 gives an overview of the definitions we gave thus far, visualizing their relationships. From these definitions, we can categorise the list of editor services presented in section 1.1. The following are passive code exploration services: *Outline*, *Code Navigation*, *Documentation*, *Diagnostic Messages*. some *Code Actions* and *Search*, while the following are categorised as active code exploration services: *Debugging* and *Testing*.

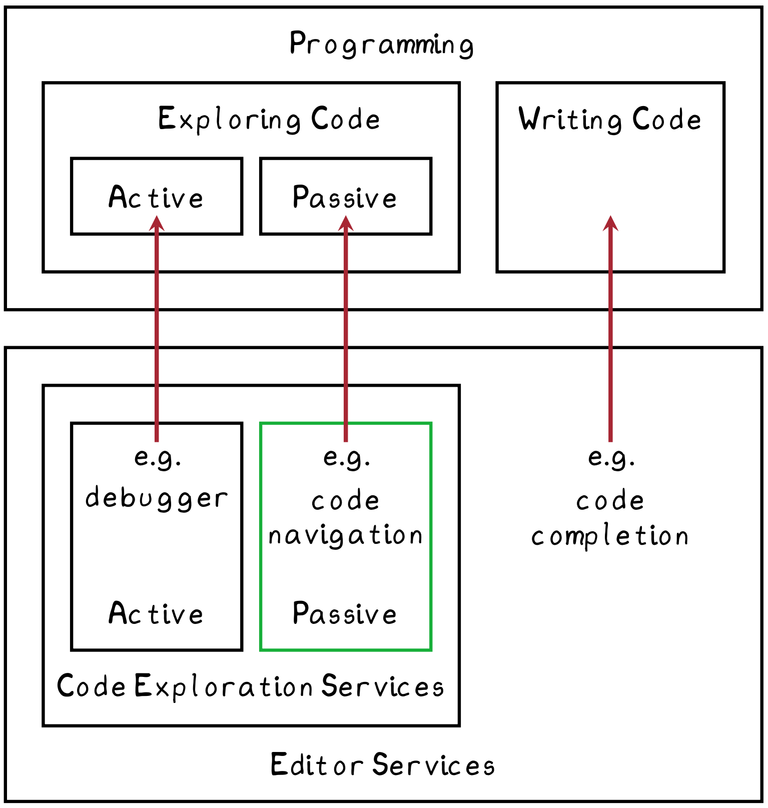


Figure 2.1: A visual overview of the definitions given in this section. Red arrows denote which kinds of services help with which kinds of programming tasks. In this thesis, we will mostly be focussing on passive code exploration services, here depicted with a green outline.

2.3 Code Exploration Services Outside Editors

Code exploration is always done in some environment, which we will call the code exploration medium. Up to now, we have mainly discussed editors as code exploration media. Editors are often a convenient code exploration media, because of the editor services many editors provide, However, editors are far from the *only* code exploration media.

For example, there are websites that display code for users to explore, such as GitHub, StackOverflow and a plethora of others. Some websites even provide some of the services that editors also provide: the category we called code exploration services in the previous section. Besides websites, PDF files (papers for example), presentation slides, books, videos and whiteboards might all be considered code exploration media as well, which just like websites can support certain code exploration services. For example, almost any medium, even those printed on paper, can have an outline or index, and syntax colouring.

Providing code exploration services in other media than editors is nothing new. Many solutions already exist today, which do exactly that: simplifying code exploration by providing code exploration services. To learn about what is possible in this space, we have made an overview of many important tools below. We chose these tools, not simply based on popularity, but also to highlight interesting or relevant examples of code exploration services in the field, to give a non-exhaustive overview of what options exist.

2.3.1 Language Specific Documentation Generators

To start off, many languages come with tools that can generate documentation. However, although 'documentation generator' suggests that they provide only a single editor service:

documentation, this is often far from true. Instead, such tools often provide several code exploration services together, like code navigation, and syntax highlighting, though sometimes in limited form. We will first look at some tools meant to serve only a single programming language or ecosystem, and then broaden our scope to tools which are more widely applicable.

The standard Rust distribution ships with a tool called Rustdoc (The Rust Programming Language 2023a). The tool generate documentation websites from Rust projects. In a sense, it can be seen as a kind of compiler. It takes Rust code annotated with special comments (so-called doc-comments), and produces an HTML file. The generated HTML contains only the documentation and public API of the code, not the code itself.

Rustdoc provides at least three code exploration services. The main ones are documentation, and an outline. Additionally, there is code navigation. All parts of the public API are clickable, Finally, there are bits of code that are displayed, mostly type signatures, which have coloured syntax.

Rustdoc is similar in many ways to tools such as Haddock for Haskell, Javadoc for Java, and countless comparable tools for other languages.

Just like Rustdoc can generate documentation websites for Rust, the Agda compiler can produce HTML output based on Agda source code. As we have seen, many languages have this capability, so although it can be useful, it is not especially novel. However, we mention Agda separately because the language also natively supports generating LaTeX for use in papers (Agda 2023). The tool mainly performs syntax colouring.

2.3.2 Language Agnostic Documentation Generators

Doxygen Doxygen (van Heesch 2022) generates documentation pages just like Rustdoc does, and therefore seems quite similar in scope at first glance. However, while Rustdoc only works for a single language: Rust, and similarly Haddock only works for Haskell, Doxygen actually supports quite a wide range of languages. For example, C, C++ and Java, and with extensions even some less common languages are supported, as long as their syntax is somewhat similar to that of C.

Furthermore, Doxygen can produce all kinds of output formats, not just webpages. Just like Agda, Doxygen supports outputting LaTeX, but also man pages and XML. The XML output is interesting, because it is not intended to be human-readable, and instead captures the structure of programs in a more machine-readable format.

To provide code navigation in the generated documentation, Doxygen needs to have a way to figure out which parts of a codebase reference one another. This is different for every language, and thus quite a complex task for a tool that aims to support many different languages. All previously mentioned documentation generator tools have it much easier in this regard: they have only a single language to deal with.

To avoid this complexity, Doxygen uses a different program called CTAGS to generate reference information (CTAGS 2023). We will discuss this tool further in section 2.3.4.

CBS For a language called Component Based Semantics (CBS), Mosses (2019) created a system to generate a documentation website based on source code. The tool is currently a language-specific tool just like ones discussed in the previous section. Furthermore, just like some of those tools, it can generate both HTML and LaTeX. However, what is different is that CBS is built with the Spoofax language workbench. In section 1.3.2, we discussed the possibilities language workbenches possess to provide editor services across a wide range of languages. and this is also the goal for CBS' documentation generator (Mosses 2023).

2.3.3 Websites

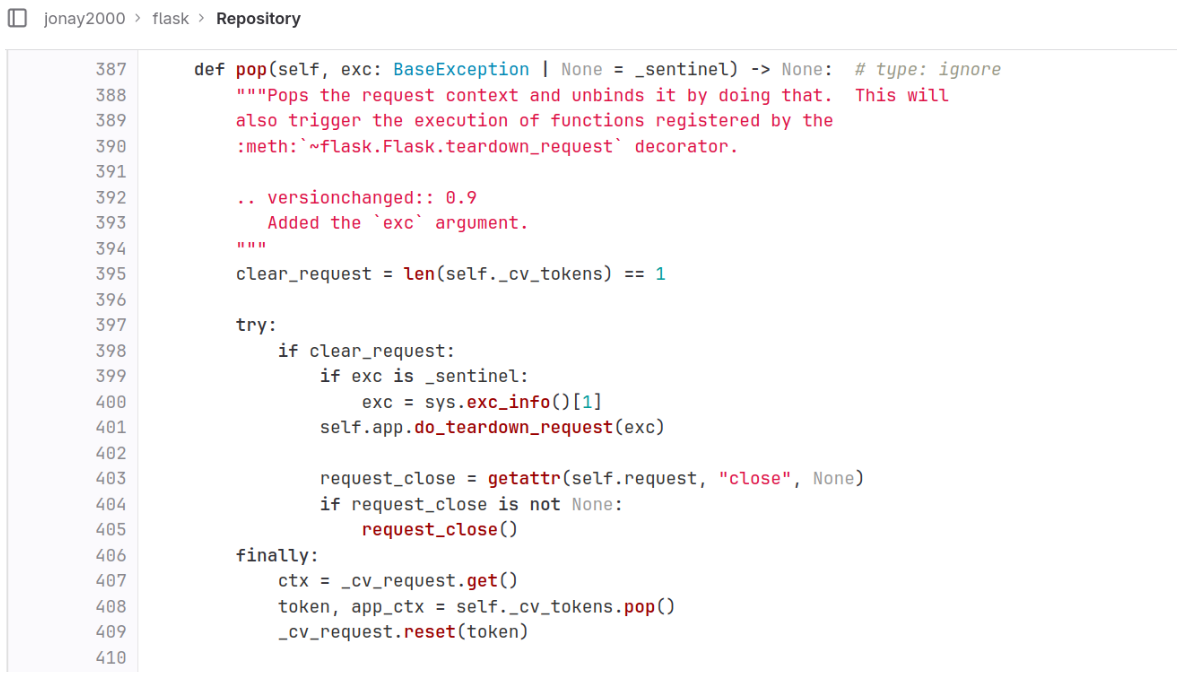
In this section, we discuss a few notable tools out of many that aid programmers collaborating while writing programs. The ones we will discuss are web-based, and display code on many pages of their website. It is not always convenient to download this code to a local development environment to be able to run and explore it. Therefore, these websites provide various online code exploration services of differing sophistication.

StackOverflow The first noteworthy tool is StackOverflow. The code exploration services StackOverflow provides are rather limited. Although many programmers look for code on the website, they essentially only provide programmers with syntax colouring, and that might just be plenty for the purposes of the site.

StackOverflow is based on questions and hand-written answers by other programmers. Answers may contain snippets of code to explain a certain concept without requiring much further context. Providing more than syntax colouring for these small snippets is likely difficult and may not even be very helpful for users.

What may happen though, is that the question that somebody asked on StackOverflow does not exactly match the one you have. As such, the given answer might not work for you. In that case, you could ask your own question, but StackOverflow also has a built-in service that finds related questions that might match your question better. One could argue that this is a form of code navigation, although not based on the contents of programs and instead based on the meaning of the code.

GitLab and GitHub In contrast to StackOverflow, GitLab is often used to store entire projects, not just snippets of code. GitLab is a platform on which people can collaborate on projects using Git, a version control system. To help people explore code on their platform, GitLab performs syntax colouring for many languages when viewing files with source code in them. Additionally, GitLab has search functionality that can be scoped to a particular project or group of projects.



```
    def pop(self, exc: BaseException | None = _sentinel) -> None:  # type: ignore
        """Pops the request context and unbinds it by doing that.  This will
        also trigger the execution of functions registered by the
        :meth:`~flask.Flask.teardown_request` decorator.

        .. versionchanged:: 0.9
           Added the `exc` argument.
        """
        clear_request = len(self._cv_tokens) == 1

        try:
            if clear_request:
                if exc is _sentinel:
                    exc = sys.exc_info()[1]
                self.app.do_teardown_request(exc)

                request_close = getattr(self.request, "close", None)
                if request_close is not None:
                    request_close()
        finally:
            ctx = _cv_request.get()
            token, app_ctx = self._cv_tokens.pop()
            _cv_request.reset(token)

```

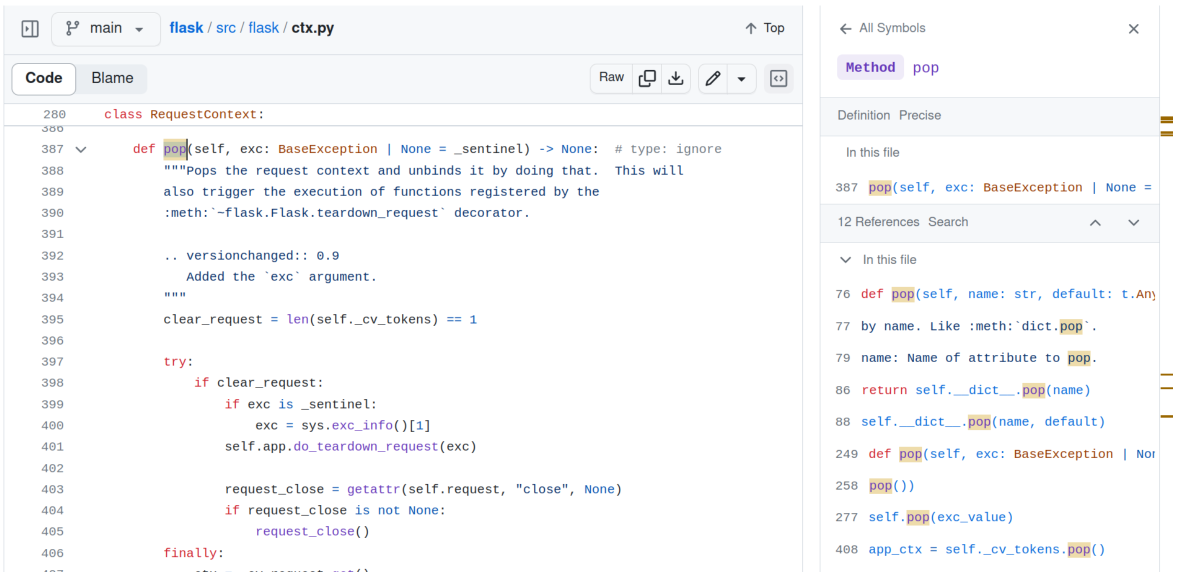
Figure 2.2: *ctx.py* from the Python Flask repository viewed on GitLab with only syntax Colouring.

GitLab provides fewer code exploration services than GitHub. Just like on GitLab, programmers often upload entire codebases to GitHub, and work on their code together through Git. However, GitHub gives us a taste some of what is possible with more advanced code exploration services.

In addition to the quite standard code colouring, GitHub allows visitors of the site to easily navigate source code of limited set programming languages (currently only Python) by jumping from usages to definitions and back, even between different files of a project. Instead of depending on exact name matches like Doxygen does, GitHub builds a model of the scoping rules of a language using Stack Graphs (Creager and van Antwerpen 2023; Github 2023b). Based on this model, they can work out which identifiers refer to which other identifiers while taking scoping rules into consideration.

In section 1.3.2, we discussed language workbenches and Spoofax. To define the scoping rules of a language in Spoofax, a model called Scope Graphs is used (Néron et al. 2015; van Antwerpen, Néron, et al. 2016; van Antwerpen, Poulsen, et al. 2018). Stack Graphs are based on Scope Graphs, but modified to be highly incremental. That means that when someone changes one file in the project, and pushes this change as part of a Git commit, GitHub does not have to re-analyse all other files in the code base.

In figure 2.3 we show a demonstration of code navigation through the Python Flask project based on Stack Graphs on Github. At the time of writing, GitHub has only enabled Stack Graphs based navigation for Python, though the plan is to provide this for more languages (Creager and van Antwerpen 2023).



```
class RequestContext:

    def pop(self, exc: BaseException | None = _sentinel) -> None:  # type: ignore
        """Pops the request context and unbinds it by doing that.  This will
        also trigger the execution of functions registered by the
        :meth:`~flask.Flask.teardown_request` decorator.

        .. versionchanged:: 0.9
           Added the `exc` argument.
        """
        clear_request = len(self._cv_tokens) == 1

        try:
            if clear_request:
                if exc is _sentinel:
                    exc = sys.exc_info()[1]
                self.app.do_teardown_request(exc)

                request_close = getattr(self.request, "close", None)
                if request_close is not None:
                    request_close()
        finally:
```

```
387 pop(self, exc: BaseException | None =
```

```
76 def pop(self, name: str, default: t.Any
77 by name. Like :meth:`dict.pop`.
79 name: Name of attribute to pop.
86 return self.__dict__.pop(name)
88 self.__dict__.pop(name, default)
249 def pop(self, exc: BaseException | Nor
258 pop())
277 self.pop(exc_value)
408 app_ctx = self._cv_tokens.pop()
```

Figure 2.3: The same *ctx.py* as shown in figure 2.2 from the Python flask repository on GitHub, showing coloured code on the left with one identifier selected: `pop`. On the right, the code navigation menu is open (which happens after an identifier is selected) showing navigation options: two in this file and 3 more in different files. Clicking these options navigate to those files in the browser. It is also possible to open a file overview on the left, to navigate between files. When no identifier is selected, the right shows an outline of the current file.

In addition to syntax colouring, code navigation (for Python) and extensive search options, GitHub users have also found custom ways to add limited diagnostics messages to code displayed on the site. For example, the Clippy check action (`https://github.com/actions-rs/clippy-check`) automatically checks code uploaded to project repositories on GitHub by using GitHub Actions, GitHub's continuous integration service, and adds comments at the places where warnings and errors are found.

2.3.4 Other tools

Finally, there are several tools which do not fit any category above, and yet are certainly worth a mention.

Web Code Colourers First, there are several JavaScript libraries which specifically solve syntax colouring on websites. For example, there are *highlight.js* (2023) and *prism.js* (2023), which essentially package a single code exploration service: code colouring, to work for hundreds of languages on any website. Several of the websites we discussed sofar use highlight.js on their website, like StackOverflow and GitLab (Kelley and StackOverflow 2020; GitLab n.d.), while GitHub uses a custom system.

LaTeX Code Colourers Next, there are similar tools that provide syntax colouring in LaTex documents. Since this thesis was written using LaTeX, we can demonstrate those in this document. First, there's the LaTeX `listings` package, which simply highlights a number of predefined keywords, as figure 2.4 shows.

```
fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 0,
        1 => 1,
        n => fibonacci(n - 1) + fibonacci(n - 2)
    }
}
```

Figure 2.4: A snippet of Rust code, formatted using the LaTeX `listings` package.

Which keywords are highlighted changes per language, and users of the library can define their own set of keywords that should be highlighted. This is different to what the LaTeX `minted` package does, which uses an external tool called Brandl, Chajdas, and Abou-Samra (2023) to perform highlighting. The result looks a lot more like what code listings commonly look like on websites, as can be seen in figure 2.5.

```
fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 0,
        1 => 1,
        n => fibonacci(n - 1) + fibonacci(n - 2)
    }
}
```

Figure 2.5: A snipped of Rust code, formatted using the latex `minted` package.

CTAGS CTAGS is a tool that reads programs and codebases and generates so-called tagfiles. Tagfiles are like a table of contents of a codebase, containing information about the location and type of all items in the codebase.

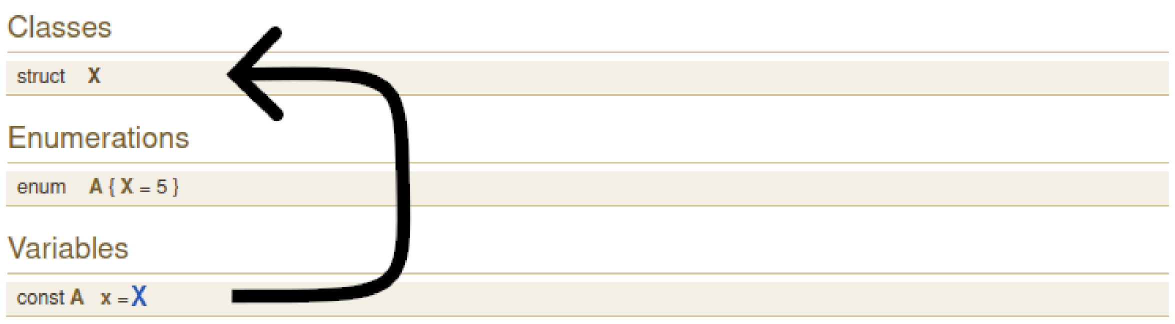
Doxygen, a tool we previously discussed in section 2.3.2, reads the output of CTAGS to provide its services. However, such an index of definitions in a codebase is useful for all kinds of tools, not just Doxygen. In fact, the main use case of tagfiles is not Doxygen, but editors to help provide editor services. Vim, for example, can read tagfiles to provide code navigation.

Although CTAGS can find identifiers in a source file, it does not understand scoping rules. The result is that tools like Doxygen, that use CTAGS to provide code navigation, sometimes make mistakes. If there are two items with the exact same name, Doxygen may refer to the wrong one or both.

Luckily, the languages that Doxygen works on have scoping rules that make triggering these mistakes somewhat difficult. For example in C++, toplevel items are not allowed to shadow each other. Still, it is possible to craft an example that shows Doxygen making mistakes in finding what parts of the program reference what other parts. We used Doxygen to generate documentation for the C++ code snippet below, in which both the enum variant `A::X` and the struct `X` are both in global scope:

```
1  #include<iostream>
2
3  enum A {
4      X = 5,
5  };
6
7  struct X {};
8  const A x = X;
9
10 int main() {
11     std::cout << x << std::endl;
12 }
```

In C++, you do not need to qualify usage of an enum variant, so on line 8, the variable `example` gets the value 5 assigned which is printed in `main`. However, Doxygen internally thinks that the enum variant is called `A::X` and the struct is just called `X`. Therefore, the `X` in `const A x = X` does not *exactly* match the enum variant name and instead Doxygen thinks this statement refers to the struct `X`, the name of which *does* match exactly. Doxygen shows this association by generating a blue link, and in figure 2.6 the arrow shows where the link (mistakenly) takes users.


Classes

struct X

Enumerations

enum A { X = 5 }

Variables

const A x = X

Figure 2.6: Doxygen links from `const A x` to `struct X` because it wrongly matches the value to the struct which is also named `X`. Doxygen should instead link to the variant named `X` in `enum A`. The arrow shows where the blue `X` link points users to.

Bootlin Elixir The Bootlin Elixir cross-referencer is an online tool to browse the source code of various large C and C++ projects such as the Linux kernel and Glibc. In Elixir, users can navigate around these large code bases easily by providing code navigation and syntax highlighting. The website is available at `https://elixir.bootlin.com`.

The cross-referencer is a custom tool written in Python, which stores metadata in an SQL database. This works for a small set of programming languages: C, C++ and assembly, as well as some formats specific to the Linux kernel such as devicetree files. The database can later be queried, which is done to generate HTML webpages for example.

Matt Godbolt's Compiler Explorer The Compiler Explorer is an online tool created by Matt Godbolt, available at `https://godbolt.org`. On the website, users can write code in a simple editor and execute it (non-interactively) on the server, and this works for a large range of different compiled programming languages. This makes the Compiler Explorer, in contrast to the other systems in this list in that it also provides active code exploration services: users can see what their code does when ran.

Furthermore, the Compiler Explorer is not just an online editor and few people would use it as such. The main purpose of the Compiler Explorer is instead to be able to choose *exactly* which compiler is used to compile the code. The website can, instead of running the code, also give the binary and assembly output of the compiler, and link each line in the source code to what assembly instructions correspond to it. With the Compiler Explorer, users can research differences in the code generated between different compilers and programming languages, and can visualise the optimisations a compiler performs.

This makes the Compiler Explorer unique in the list of code exploration systems in this chapter: it is the only tool that performs services which even sophisticated IDEs cannot provide. Few developers have the hundreds of versions of different compilers of different languages installed on their local machine, even though that would be required for an IDE to provide the features of the Compiler Explorer. Instead, providing this tool as-a-service, on remote machines which do have all these different compilers installed is much more convenient.

2.3.5 Evaluation

In this section we have discussed many examples of systems that provide a wide range of code exploration services in several media. However, many of the discussed approaches are quite narrow in scope. To make that notion more exact, we say that a system is narrow when it has one of the following shortcomings:

1. It supports only a single or small number of programming languages,

2. It supports only a single or small number of code exploration media,

3. It supports only a single or small number of code exploration services.

All the systems we looked at have at least one of these three properties. For example, the documentation generators we discussed, like rustdoc, only support a single language. The various code colourers we mention almost by-definition only support a single code exploration service. Finally, GitHub's Stack Graphs currently supports only a single language, only powers a single code exploration service and is only available in a single medium: on GitHub.

What we see here is, in essence, similar to the IDE portability problem described in section 1.3. There are m languages for which services are provided in n places and no party can feasibly be responsible for those $m \times n$ implementations. Instead, many narrow solutions form the landscape of code exploration services in various media. We call this, the $m \times n$ problem for code exploration services.

The opposite of a system that is narrow in scope is a system that is broad in scope. A broad tool is one that does have the ability to support multiple programming languages, multiple media and multiple code exploration services, and is extensible in all these directions. Although there are currently no broad solutions for code exploration services, we have already discussed one for editor services in section 1.3.1. LSP is a broad solution to the IDE Portability Problem. Through LSP, multiple editor services can be provided, in any supported editor and for any supported language.

Of course, the caveat there is that the editor or language does need to be supported. However, to support more languages or editors, LSP does not fundamentally need to change. This is an important property a broad system should have: extensibility.

The rest of this thesis will be about this: creating a broad solution that solves the $m \times n$ problem for code exploration services. A solution which can support multiple kinds of code exploration services, in a language-agnostic manner, in multiple code exploration media.

Chapter 3

The Design of the Codex Format

In the previous chapter we identified a problem analogous to the IDE portability problem: the $m \times n$ problem for code exploration services. Current systems that provide code exploration services are all narrow in scope which hinders reuse causing this $m \times n$ complexity. A solution for this problem is a system that is not narrow, but broad in scope and that can support multiple languages, multiple code exploration media, multiple code exploration services and which is extensible in all these dimensions.

The solution we present in this chapter is the Codex metadata format, which forms the foundation behind Codex, a broad and extensible system for code exploration services which we will demonstrate in chapter 4. The metadata format is to codex what the language server *communication protocol* is to LSP: it connects language-specific tools to language-agnostic tools. This is the main reason both LSP and Codex are broad in scope. In figure 3.1 we show the Codex metadata format, and the rest of this chapter is dedicated to our reasoning for why the format is what it is.

3.1 The Codex Metadata Format

The Codex metadata format needs to be able to store metadata for all code exploration services as described in section 2.1. To do so, the format needs to represent:

- references to specific sections of the source code, which aids code navigation and relating structures in the outline;
- classifications of certain parts of source code, for example to indicate the syntactic classification of a token or the semantics of a structure;
- extra information relating 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, there is a difference between references from a definition of an item to a usage, and from a usage back to a definition, which we want to be able to represent for code navigation.

In the Codex metadata format, we define three different data types to represent these three kinds of metadata, which can be seen in figure 3.1. References are made using the `Span` data type representing a source locations as two numbers and a string: a start in number of unicode codepoints, a length and a file path relative to the root of the project. Classifications are made with the `Classification` data type, and free-form data is represented as a `String`.

```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)>;
```

Figure 3.1: 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 chapter 4.

Using combinations of these three data types, the metadata required for different code exploration services can be stored. The `Relation` type in figure 3.1 represents these combinations of more primitive types. There are four kinds of `Relations` defined, for four different code exploration services.

Finally, Codex metadata consists of many such relations. A single source file likely contains many references between locations, and many tokens which are classified using `Syntax` relations. Each `Relation` is paired with a source location, also called a `Span`, and stored in a large list which is called `Metadata` in figure 3.1.

A small example of what some Codex metadata might look like when using this format can be found in figure 3.2. The example shows samples of each kind of metadata generated from a Rust file in a serialized representation as JSON. How we generated this exactly will become clearer in chapter 4.

3.2 Language Agnostic Classification

It is important that the way this metadata is stored is in no way tied to any specific programming language. If that were the case, we would unnecessarily limit the scope of Codex, just like the narrow tools we discussed in section 2.3.

```
["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"]
  }
}]
```

Figure 3.2: An excerpt of serialized metadata in the codex format, showing what data for the different kinds of metadata categories defined in section 3.1 look like. The metadata was generated by taking the source code in figure 3.1 and running it through the Codex tool we show in chapter 4. Out of about a hundred lines of metadata, only a few lines are shown.

Despite the success of LSP, one could argue that this is a flaw in LSP. In LSP, a symbol can fall into one of a limited set of categories, which is the symbol's `SymbolKind` (Microsoft 2022b). It is not difficult to find languages with symbol kinds that do not fit into these limited number of categories: Rust has structs but a struct is not a valid `SymbolKind`, and Haskell's typeclasses fail to fit in as well. To represent these items, we could approximate: a struct might become a class, and a typeclass an interface. However, this does not always work. C++ has both classes and structs, making it ambiguous if we labelled these both as classes.

The approach LSP takes, with a limited number of categories into which all languages must fit is doomed to fail. Because of the wide variety of programming languages in existence, and subtle differences in meaning of language constructs between languages, we cannot make an exhaustive list of syntactical categories which applies to every language there is.

3.2.1 Hierarchical Categorisation

An approach that avoids making such an exhaustive list was first used for TextMate, an editor for macOS (MacroMates Ltd. 2021). This approach is now used in facilitating syntax highlighters in multiple editors, like Visual Studio Code, all Jetbrains editors, and Atom.

TextMate uses custom syntax specifications of languages in the form of regular-expression-based grammars in order to support syntax colouring. Such grammars label the different syntactical elements. Colour themes can then apply different colours to items with specific labels. Because the people who make grammars are not necessarily the same as the people making the themes, a standardised labelling system had to be invented such that all themes could be compatible with all grammars. That allows anyone to make their own themes.

Instead of making an exhaustive list, TextMate uses a hierarchical approach, which is not so different from taxonomies in biology. In TextMate, an equals sign in a variable assignment in Rust might be labelled `keyword.operator.assignment.rust` using this hierarchical categorisation scheme. The dots in this label separate parts of the label, going from generic to specific. First of all, the equals sign is a keyword, but more specifically an operator, specifically meant for assignment, specifically in Rust.

Now, TextMate themes can match on these labels in a similar way to how CSS classes match: if a theme only knows how to colour keywords, this assignment operator gets the same colour as all keywords. However, a theme can also specify a specific colour for all items labelled `keyword.operator`, or even more specific. The advantage is that this system can both capture all the language-specific details of syntax, as well as being easy to read for tools which might not be aware of these language-specific details.

In the Codex metadata format, to be completely language-agnostic, it makes sense to make any language-specific labelling such as syntactical categories hierarchical. In fact, as figure 3.1 shows, we use classifications for every relation.

3.3 Generating and Using Metadata

Using the Codex metadata format, metadata can be generated and stored in a language-agnostic way. The question then becomes, who or what generates the metadata? By keeping the format simple, a wide range of tools could be *metadata generators*.

Many of the narrow tools we discussed in section 2.3 can be metadata generators just by translating their output. For example, CTAGS could be a metadata generator. Currently, CTAGS outputs tags files, but these simply contain encoded metadata for a single source file. Similarly, we could imagine GitHub's Stack Graphs to be a generator of metadata, though mainly for reference information.

A completely different example of a possible metadata generator is a compiler. Naturally, a compiler has a very good understanding of the program it is compiling. To generate machine code for a program, a compiler already has to parse the source and do an analysis of references. A compiler could, apart from outputting machine code, also output some Codex metadata.

Through the Codex metadata format, a wide range of these *metadata generators* can be connected to what we call language agnostic *presentation generators* that can take metadata and source code to create rich presentations of that source code is various media. Presentation generators are the application what Codex is designed for, though one can imagine tools with other purposes reading Codex metadata to gain information about some program.

The concept of metadata generators and presentation generators is similar to how LSP functions. They call metadata generators *language servers* and the presentation generators are analogous to *editors*. With LSP, the two sides communicate over the language server *protocol*, which is comparable to the Codex metadata format.

3.4 A Format Instead of a Protocol

An important difference between the Codex metadata format and LSP is that the former is a data *format*, while the latter is a *protocol*. That is because there is a large difference between editors and presentation generators.

Editors using LSP are lazy; information is *only* sent when requested by the user through the editor. That is possible because editors are also very *dynamic*: they can easily run arbitrary code when the user gives inputs, to make the necessary requests for information. This can be beneficial for performance, not all information needs to be available from the start, and is especially necessary because source code in editors is constantly changed, so metadata very quickly becomes outdated.

Code presentations in other media cannot work this way. Under LSP, editors are both presentations, and also the tools that interpret and present metadata received the metadata from a language server. However, these roles become separated in other media, because other media are not as dynamic and flexible as editors are.

Take PDF documents as an example. While an editor can dynamically make requests to a language server running in the background to get up-to-date information about the program which is written, we cannot make requests like that inside a PDF document. We illustrate this in figure 3.3.

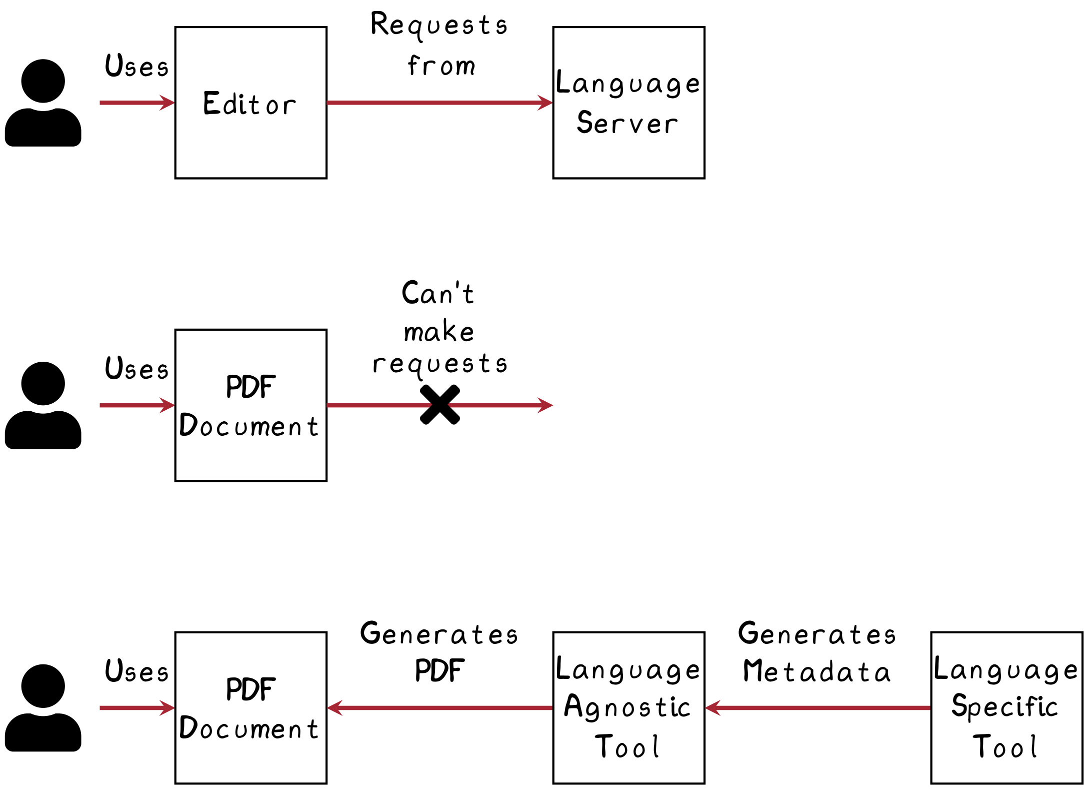


Figure 3.3: Although a language-agnostic editor can directly make requests to a language-specific source of information (as is the case with LSP), this architecture cannot work in code exploration media where making requests are impossible such as in PDF documents. Code exploration services must instead be embedded in such media whenever they are generated.

Instead, if we want to provide code exploration services in a PDF document, those need to be embedded into the PDF itself at the moment the PDF is generated. This baking-in process may involve colouring some text in the document where source code is presented, and, for example, adding extra hyperlinks to navigate between different pieces of source code. This embedding is done by the presentation generator, making *it* analogous to an editor with the capability to access language-specific metadata. In that way, the presentation generated by the presentation generator has the possibility to be entirely static.

Because the entire presentation is generated at the same time, all metadata is needed at the same time. Therefore, a request-response model like LSP makes little sense, in the end all data will be requested. Instead, a method of communication in which all information is packaged together can be used: a data format instead of a protocol. We foresee several advantages of doing this.

First, there is no need for two-way communication between the metadata generator and presentation generator. Instead, all data flows one way. The metadata generator can generate all metadata at once, then the presentation generator can interpret all the metadata at once.

Next, after the metadata is generated, the presentation generator does not have to interpret the metadata immediately. Instead, the metadata could be written to a file and used later. One could imagine metadata being committed along with programs to a Git repository, or being generated by a continuous integration pipeline. Then, other parties could use the metadata to generate their own presentations, also after the tooling that generated the metadata is maybe not available anymore. Git providers like GitHub or GitLab could even use the metadata to present code to users automatically, regardless of the language (as long as the authors of the code also provided the metadata).

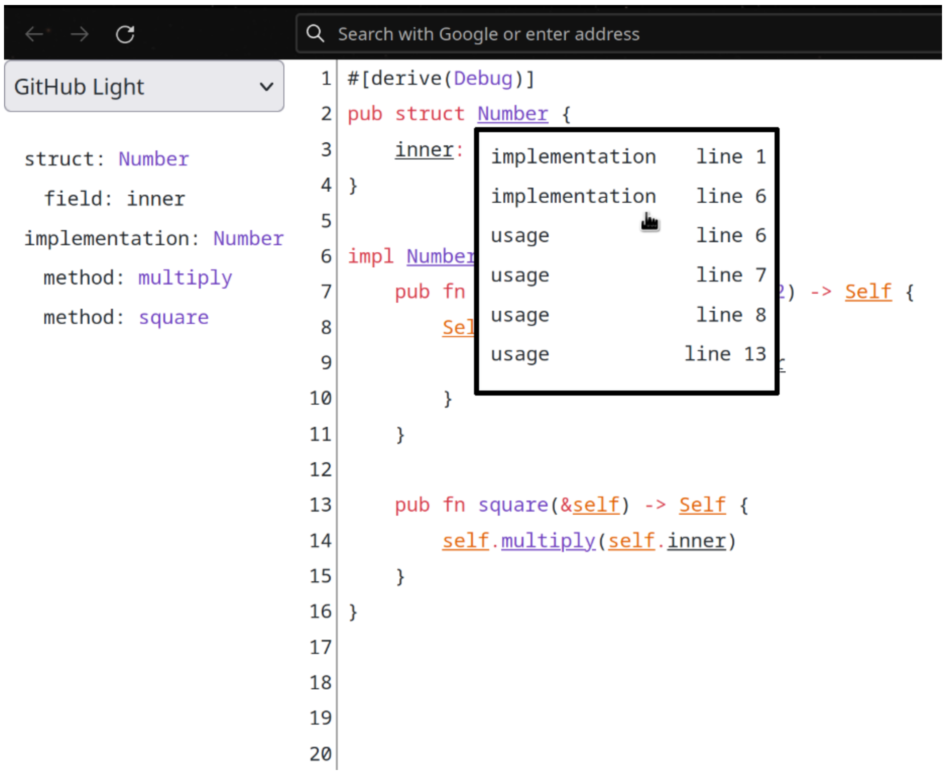
Lastly, we can imagine the stored metadata serving as a kind of cache. Before metadata generators start, they could first read the old metadata from a file, and then only generate new metadata for code that has changed, updating the cache.

Chapter 4

Demonstration

In the previous chapter, we described the design of the Codex metadata format. Using that format, we claimed we can decouple language-specific metadata generators from language-agnostic code exploration media and make a broad and extensible solution to the $m \times n$ problem for code exploration services. The question now is, can the Codex metadata format indeed do that?

In this chapter we demonstrate that it does by looking at the prototype system we built which we call Codex. Codex currently demonstrates four different language-specific tools that generate program metadata which communicate, using the Codex metadata format, with two different language-agnostic systems that generate presentations of code in two different media.



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

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

impl Number
    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
```

Figure 4.1: A fragment of Rust code presented in an interactive HTML webpage. The colour information, reference information (underlines), outline, and warnings are derived from metadata stored in the Codex metadata 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 webbrowser to the relevant code.

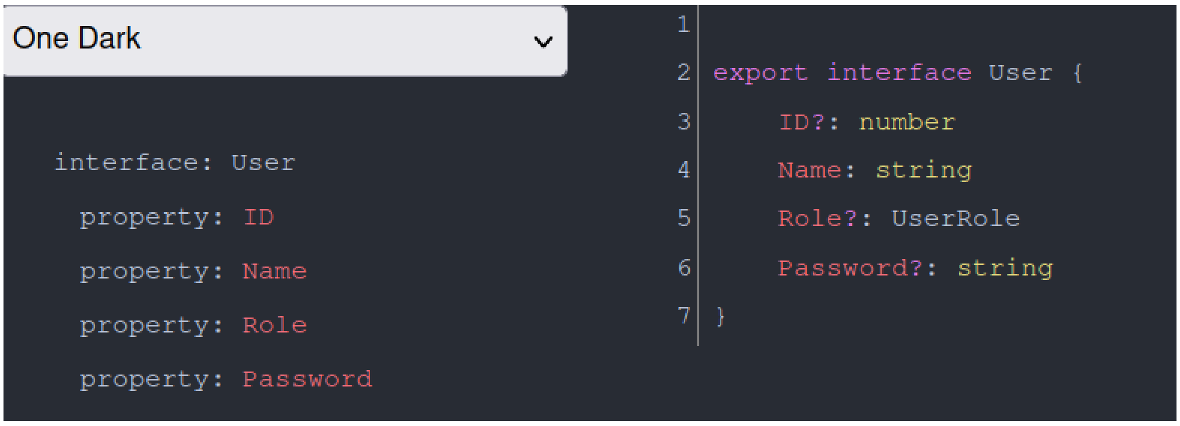
The source code of this tool can be found online at `https://github.com/jdonszelmann/codex`.

4.1 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 visualisations. Many websites such as StackOverflow, GitHub and GitLab can present users' source code, sometimes with some basic code exploration services.

Using the Codex prototype, we can produce HTML documents from source code and its corresponding Codex metadata. What programming language the source code was written in does not matter, as long as there is associated metadata. In figure 4.1 we show an example of Codex's presentation of a source file containing Rust code in a web browser. In the generated HTML document, we implemented several code exploration services: syntax colouring, 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 visualisation. Instead of assigning colours to tokens, we add CSS classes based on the token's classification (See section 3.2 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. In figure 4.2 we show a presentation of a program in HTML, formatted with a different colour scheme.


One Dark

```
interface: User
  property: ID
  property: Name
  property: Role
  property: Password
```

```
export interface User {
    ID?: number
    Name: string
    Role?: UserRole
    Password?: string
}
```

Figure 4.2: A fragment of TypeScript code presented in an interactive HTML webpage in a different colour scheme than figure 4.1. No code navigation is available in this example, since this information is derived from LSP in other languages, and TypeScript has no good LSP available. Editors like Visual Studio Code instead have TypeScript support built-in and derive their editor services for TypeScript from `tsc` directly.

4.2 Presenting Code in PDF Documents through LaTeX

A different presentation of the same Rust code of figure 4.1 is shown in figure 4.3 embedded into this thesis'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 visualisations 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 4.4 shows source code from a different file in the same Rust project as the example in figure 4.1. The two files reference each other, and when reading this thesis 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 figure 4.3, figure 4.4 and in figure 4.5.

Because both the HTML presentation and this LaTeX presentation rely solely on metadata in the Codex format, both are completely language-agnostic. Although figure 3.2 is mainly meant to demonstrate what the Codex metadata format looks like when it is serialised, the syntax colouring in the example itself is created using Codex. Therefore, figure 3.2 is also a demonstration of how, through the Codex metadata format, we can present code written in different languages.

```
#[derive(Debug)]
pub struct Number^{i,i,u} {
    inner: u32,
}

impl Number^{d,i,u} {
    pub fn multiply(&self, other: u32) -> Self {
        Self {
            inner: self.inner * other
        }
    }

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

Figure 4.3: The same fragment of Rust code as shown in figure 4.1, but embedded in this thesis document through LaTeX source code generated from the Codex metadata format. When this document's PDF is viewed digitally, some elements such as variables are clickable and navigate to the referenced location. When an item is referenced more than once, superscript annotations are inserted that facilitate navigation: 'd' means definition, 'i' means implementation and 'u' means usage.

4.3 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

```
fn print_square(n: Number^{d,i,u,u}) {
    println!("{:?}", n.square());
}
```

Figure 4.4: This is the continuation of the example Rust code in figure 4.3. Code navigation enables quick navigation between the two pieces of code on different pages, when reading this thesis' PDF digitally.

one originally used by the TextMate editor (MacroMates Ltd. 2021). Grammar definitions for many languages, both small and large, 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 3.2.1 how we use this function for syntax colouring and other code exploration services in the Codex format. Because we use this system for syntax colouring, translating from TextMate output to the Codex format is very simple. TextMate grammar files already contain such classifiers, so we can include 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 colour 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. You can find this implementation at code-exploration-services-lib/src/textmate

CTAGS CTAGS (2023) 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 accurately derive code navigation from its output alone. Despite Doxygen doing this, as discussed in section 2.3, name shadowing easily fools the primitive name resolution CTAGS performs. However, CTAGS' output can be used to generate an outlines for source files, which is what Codex does. The implementation of this tool can be found in code-exploration-services-lib/src/input/subsystems/ctags. An example of such an outline can be found in figure 4.1.

The Language Server Protocol In section 1.3.1, we already mentioned LSP (Microsoft 2022a) in the context of the $m \times n$ problem. Although LSP is a protocol 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, where it requires 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 want to analyse. Then we query this language server 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 for figures 3.1, 4.1 and 4.3 is generated.

4.4 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 visualisations of code that Codex can gener-

```
use std;

effect Abort {
  abort() a
}

let hAbortu,u = fn(default) {
  handler {
    return(x) { x }
    abort() { default }
  }
};

let safe_divu,u = fn(x, yu,u) <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 4.5: 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 colouring and code navigation in this example.

ate 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 (Poulsen and van der Rest 2023; Diepraam 2023). 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 outputs 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 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 thesis, 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 figure 4.5.

What's interesting to note is that the Codex metadata format has no problem supporting reference resolution for effects and elaborations, a language feature not common among larger programming languages. When Elaine outputs metadata, elaboration references get the label `definition.elaboration` (see figure 3.1), allowing consumers of the metadata to visualise it.

4.5 Evaluation

In this chapter, we demonstrated Codex, a prototype broad solution for language-agnostic code exploration services. We showed that with Codex, we can generate presentations of source code written in several programming languages, based on metadata gathered from a diverse set of language-specific tools, which was then stored in the Codex metadata format. We also showed that Codex is extensible in this regard: we can create new generators of metadata like the one we created for Elaine.

We showed that we can make presentations of source code, with code exploration services, in two different media: Websites and PDF documents. The prototype presentation generators for these media are completely language-agnostic, and in this chapter we have shown examples of Codex making presentations of Rust, TypeScript, Elaine, as well as JSON in the previous chapter in figure 3.2.

The Codex metadata format is also extensible in what metadata it can store. By adding different relations to the format as specified in section 3.1, metadata for different purposes can be added.

With that, we demonstrate that Codex is not narrow in any of the three categories highlighted in section 2.3.5:

1. Codex is extensible to support any number of programming languages through the language-agnostic Codex metadata format,

2. Codex is extensible to support any code exploration medium that can interpret the Cdoex metadata format,

3. The Codex metadata format is extensible to store metadata for any passive code exploration service.

Chapter 5

Considerations and Tradeoffs

In chapter 4, we demonstrated that the Codex metadata format indeed has the capability to decouple language agnostic presentation generators from language-specific metadata generators. Still, the Codex metadata format is far from ideal, which we experienced while designing the metadata and presentation generators that use the Codex metadata format. In this chapter, we will discuss these findings.

5.1 Storing Locations in Source Code

In the Codex metadata format we are describing metadata about parts of source code. Therefore, we need a way to refer to specific locations in the source files to connect metadata to those places. In the Codex metadata format, we represent this with the `Span` type, as we show in section 3.1.

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⟩. For example, the Rust compiler stores offsets from the start of the document, while LSP stores pairs.

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 chapter 6).

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, though this problem can be mitigated with an index mapping line numbers to byte offsets. 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.

5.1.1 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, specifies offsets to be given as a number of UTF-16 characters by default. 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, Codex started out being byte-offset based which caused issues regularly. Using UTF-8 did come at the cost of some performance: in Codex, every time we want to know the text of a source file at a certain offset, we need to traverse the entire file, and the current implementation does not keep an index.

5.2 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. In the Codex format, we take the latter approach as was outlined in section 3.1.

5.2.1 Multi-part spans

One thing to keep in mind is that spans are not always contiguous in all programming languages. Agda[1], 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.

5.3 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 achieve this, 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.

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

Referencing locations can be done with any of the techniques discussed in section 5.1. 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 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.

5.3.1 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.

5.4 Accuracy

An interesting aspect of the Codex metadata format is how accurately it represents its information. The goal of the Codex metadata format is to connect different tools with each other. One or more language-specific tools generate metadata, and the stored metadata can then later be used to create presentations.

The presentation generators have to assume that all metadata is correct. Because they are language-agnostic, they cannot do much else. Therefore, the accuracy of Codex entirely depends on the accuracy of the tools generating the metadata over which we might have little control. We have seen this go wrong a few times, where the tools we depend disagree on the location of metadata. One tool might include the spaces around a token where another does not.

One way this problem could be partially mitigated, is to first tokenize the source using a language-specific tokenizer. Instead of relating metadata to arbitrary spans of text, we can

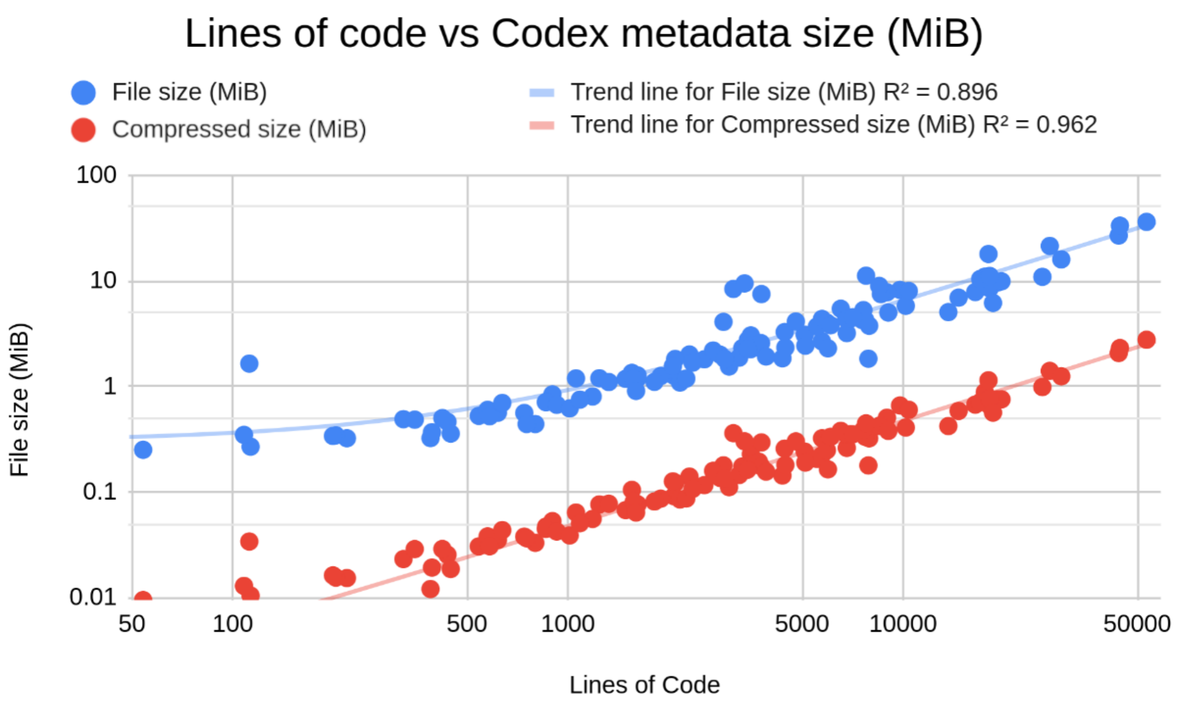


Figure 5.1: To give an indication for the size metadata files grow to, we used our proof of concept to generate metadata for the 100 most downloaded Rust projects. We chose Rust because for this language, our implementation currently generates the most metadata. In the plot, we compare the sizes of the 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).

relate metadata to a specific token. When a tool generates metadata at a location that is included by a token, the entire token might get this metadata.

5.5 Metadata Size

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.

The size of the stored metadata is no primary concern to us: our goal with Codex was to show the possibility of a language agnostic metadata format connecting language-specific metadata generators to language-agnostic presentations. However, if it turns out that even for small projects we produce an unmanageable amount of metadata, our proof of concept 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[2], and ran our generator on each of them. These projects ranged from a few tens of lines of code to thousands, comments removed. All four types of metadata relations which our proof of concept currently supports were generated for these 100 projects.

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

In figure 5.1, 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 algorithm, Zstd Meta Platforms, Inc. 2023, 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 algorithm 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.

This page was unintentionally left blank.

Chapter 6

Future Work

Although we show that the Codex metadata format can successfully decouple language-specific metadata generators from language-agnostic presentations of source code, Codex is no perfect solution yet. For example, although we theorise that many passive *code actions* can be encoded in Codex through an extra relation, we did not demonstrate this in our prototype. Furthermore, the current definition of the Codex does not support the kind of multi-part spans discussed in section 5.2.

Apart from these specific changes to the Codex metadata format and Codex prototype, there are also some larger unanswered questions The first is to see whether is it possible to encode and provide certain guarantees about the metadata 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 as well.

Next, although Codex is not intended to update metadata in real-time like LSP, sometimes programs do change and metadata is regenerated. The Codex prototype demonstrated in this thesis takes quite some time to gather metadata for a whole project, especially when a language server is queried. One way to improve this is to first figure out which parts of a code base changed, to avoid generating metadata for an entire code base. Codex could first read old metadata files, and then update only the sections that change in an incremental fashion.

Lastly, but most importantly, is an issue of standardisation. Although we showed that the Codex metadata format is capable of decoupling language-specific tools from language-agnostic tools, the only real way in which it can work if its widely supported. On could imagine GitHub reading Codex metadata files in the root of a repository and basing its navigation on it, or various compilers outputting metadata on their own. The only way that that is possible is if everyone agrees to use the same format, whether that is the Codex format or another.

Conclusion

Code exploration is an important part of programming. Many text editors and IDEs provide *editor services*, or more specifically *code exploration services* that can help programmers explore code. However, although programmers also regularly explore code in other coed exploration media such as on websites and in PDF documents, we find that only few code exploration services are supported there.

Existing tools that do provide code exploration services in other media are narrow in scope: they are not extensible in which programming languages, code exploration services and code exploration media they support. We established that a problem analogous to the IDE portability problem exists when providing code exploration services outside of editors: the $m \times n$ problem for code exploration services. Thus, we set out to create a broad solution for code exploration services, one that decouples the supported programming languages from the code exploration media in which the programs are explored.

Our solution, the Codex metadata format, is a language-agnostic format for describing the metadata of a code base, with the purpose of providing code exploration services in media such as websites and PDF documents. The format decouples generating the metadata from its presentation, addressing the $m \times n$ problem. 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.

Bibliography

Agda (2023). *Generating LaTeX*. URL: https://web.archive.org/web/20230821093901/https://agda.readthedocs.io/en/v2.6.3.20230805/tools/generating-latex.html.

Allamanis, Miltiadis et al. (2018). "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Computing Surveys* 51.4. DOI: 10.1145/3212695. URL: https://doi.org/10.1145/3212695.

Ben-Nun, Tal, Alice Shoshana Jakobovits, and Torsten Hoefler (2018). "Neural Code Comprehension: A Learnable Representation of Code Semantics". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. Ed. by Samy Bengio et al., pp. 3589–3601. URL: http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.

Brandl, Georg, Matthäus Chajdas, and Jean Abou-Samra (Sept. 10, 2023). *Pygments*. URL: https://pygments.org/.

Carbonnelle, Pierre (2023). *Top IDE index*. URL: https://pypl.github.io/IDE.html (visited on 06/28/2023).

Creager, Douglas A. and Hendrik van Antwerpen (2023). "Stack Graphs: Name Resolution at Scale". In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASIcs.EVCS.2023.8. URL: https://doi.org/10.4230/OASIcs.EVCS.2023.8.

CTAGS, Universal (2023). *ctags*. URL: https://github.com/universal-ctags/ctags (visited on 06/27/2023).

Diepraam, Terts (2023). "Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature".

Dönszelmann, Jonathan, Daniël A. A. Pelsmaeker, and Danny M. Groenewegen (Oct. 2023). "Codex: a Metadata Format for Rich Code Exploration".

Eclipse Foundation (2023a). *Eclipse Che™*. URL: https://www.eclipse.org/che/.

— (2023b). *Theia*. URL: https://theia-ide.org/.

Erdweg, Sebastian, Tijs van der Storm, Markus Völte, et al. (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*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_11. URL: http://dx.doi.org/10.1007/978-3-319-02654-1_11.

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, et al. (2015). "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer*

Languages, Systems & Structures 44, pp. 24–47. doi: 10.1016/j.cl.2015.08.007. url: http://dx.doi.org/10.1016/j.cl.2015.08.007.

Eysholdt, Moritz and Heiko Behrens (2010). "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309.

Fowler, Martin (2005). *Language Workbenches: The Killer-App for Domain Specific Languages?* url: http://www.martinfowler.com/articles/languageWorkbench.html.

Github (2023a). *Your AI pair programmer*. url: url (visited on 05/01/2023).

— (2023b). *stack-graphs*. url: https://github.com/github/stack-graphs/ (visited on 09/08/2023).

GitHub Copilot litigation (Nov. 1, 2022). url: https://web.archive.org/web/20230815201014/https://githubcopilotlitigation.com/ (visited on 08/15/2023).

GitLab (n.d.). *Syntax Highlighting*. url: https://web.archive.org/web/20230802094534/https://docs.gitlab.com/ee/user/project/highlighting.html.

highlight.js (2023). url: https://highlightjs.org/ (visited on 09/09/2023).

IEEE/The Open Group (2017). *ed(1p)*. url: https://man7.org/linux/man-pages/man1/ed.1p.html.

JetBrains (Feb. 3, 2022). *Structural search and replace*. url: https://web.archive.org/web/20230314140106/https://www.jetbrains.com/help/idea/structural-search-and-replace.html.

JSON-RPC working group (Jan. 4, 2013). *JSON-RPC 2.0 Specification*. url: https://www.jsonrpc.org/specification (visited on 03/28/2023).

Kalleberg, Karl Trygve and Eelco Visser (Mar. 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. Braga, Portugal: Elsevier. url: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-018.pdf.

Kats, Lennart C. L., Karl Trygve Kalleberg, and Eelco Visser (Apr. 2008). "Generating Editors for Embedded Languages. Integrating SGLR into IMP". In: *Proceedings of the Eight Workshop on Language Descriptions, Tools, and Applications*. Ed. by Jurgen J. Vinju and Adrian Johnstone. Vol. 238. Electronic Notes in Theoretical Computer Science 5. Elsevier.

Kats, Lennart C. L. 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*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, pp. 237–238. isbn: 978-1-4503-0240-1. doi: 10.1145/1869542.1869592. url: http://doi.acm.org/10.1145/1869542.1869592.

Katzan Jr., Harry (1969). "Batch, conversational, and incremental compilers". In: *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1969 Spring Joint Computer Conference, Boston, MA, USA, May 14-16, 1969*. Vol. 34. AFIPS Conference Proceedings. AFIPS Press, pp. 47–56. doi: 10.1145/1476793.1476813. url: http://doi.acm.org/10.1145/1476793.1476813.

Keidel, Sven, 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*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, pp. 152–162. isbn: 978-1-4503-4447-0. url: http://dl.acm.org/citation.cfm?id=2997368.

Kelley, Ben and StackOverflow (Sept. 24, 2020). *Goodbye, Prettify. Hello highlight.js! Swapping out our Syntax Highlighter*. url: https://web.archive.org/web/20221001151631/https://meta.stackexchange.com/questions/353983/goodbye-prettify-hello-highlight-js-swapping-out-our-syntax-highlighter?cb=1.

Kladov, Alex (Apr. 25, 2022). *Why Lsp?* url: https://matklad.github.io/2022/04/25/why-lsp.html.

MacroMates Ltd. (2021). *TextMate for macOS*. URL: `https://macromates.com/`.

Martin, Robert C. (2009). *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall. ISBN: 978-0-13-235088-4. URL: `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0132350882,00.html`.

Meta Platforms, Inc. (2023). *Zstandard*. URL: `https://facebook.github.io/zstd/#other-languages`.

Microsoft (2022a). *Language Server Protocol*. URL: `https://web.archive.org/web/20230310034722/https://microsoft.github.io/language-server-protocol/` (visited on 03/10/2023).

— (Oct. 5, 2022b). *SymbolKind*. URL: `https://web.archive.org/web/20230318082328/https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#symbolKind`.

Mosses, Peter D. (2019). "Software meta-language engineering and CBS". In: *Journal of Computer Languages* 50, pp. 39–48. DOI: `10.1016/j.jvlc.2018.11.003`. URL: `https://doi.org/10.1016/j.jvlc.2018.11.003`.

— (2023). "Using Spoofax to Support Online Code Navigation". In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ISBN: 978-3-95977-267-9. DOI: `10.4230/OASIcs.EVCS.2023.21`. URL: `https://doi.org/10.4230/OASIcs.EVCS.2023.21`.

Néron, Pierre et al. (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*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, pp. 205–231. ISBN: 978-3-662-46668-1. DOI: `10.1007/978-3-662-46669-8_9`. URL: `http://dx.doi.org/10.1007/978-3-662-46669-8_9`.

Pelsmaeker, Daniël A. A. (2018). "Portable Editor Services". MA thesis.

Poulsen, Casper Bach and Cas van der Rest (Jan. 2023). "Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects". In: *Proceedings of the ACM on Programming Languages* 7.POPL, pp. 1801–1831. DOI: `10.1145/3571255`. URL: `https://doi.org/10.1145/3571255`.

prism.js (2023). URL: `https://prismjs.com/` (visited on 09/09/2023).

Sourcegraph (2023). *A community driven source of knowledge for Language Server Protocol implementations*. URL: `https://web.archive.org/web/20230418003505/https://langserver.org/` (visited on 05/02/2023).

The Rust Programming Language (2023a). *The Rustdoc Book*. URL: `https://web.archive.org/web/20230314235749/https://doc.rust-lang.org/rustdoc/`.

— (2023b). *rustfmt*. URL: `https://github.com/rust-lang/rustfmt` (visited on 08/18/2023).

Thompson, Ken (1969). *The ed line editor*.

Van der Storm, Tijs (2011). "The Rascal language workbench". In: *CWI. Software Engineering [SEN]* 13, p. 14.

Van Antwerpen, Hendrik, Pierre Néron, et al. (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*. Ed. by Martin Erwig and Tiark Rompf. ACM, pp. 49–60. ISBN: 978-1-4503-4097-7. DOI: `10.1145/2847538.2847543`. URL: `http://doi.acm.org/10.1145/2847538.2847543`.

Van Antwerpen, Hendrik, Casper Bach Poulsen, et al. (2018). "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. DOI: `10.1145/3276484`. URL: `https://doi.org/10.1145/3276484`.

Van Heesch, Dimitri (2022). *Doxygen*. URL: `https://web.archive.org/web/20230819152523/https://www.doxygen.nl/index.html` (visited on 08/19/2023).

Van Rossum, Guido, Barry Warsaw, and Nick Coghlan (July 5, 2001). *Style Guide for Python Code*. URL: `https://peps.python.org/pep-0008/`.

Acronyms

AI Artificial Intelligence

AESI Adaptable Editor Services Interface

ASCII American Standard Code for Information Interchange

AST Abstract Syntax Tree

DSL Domain-Specific Language

GUI Graphical User Interface

IDE Integrated Development Environment

JSON JavaScript Object Notation

JSON-RPC JavaScript Object Notation (JSON) Remote Procedure Call (JSON-RPC working group 2013)

LSP Language Server Protocol

PDF Portable Document Format

SLE Software Language Engineering

Appendix A

Paper

As part of this thesis, a paper was submitted to the Software Language Engineering (SLE) conference. Unfortunately it was not accepted, in part due to the positioning and the size of the related work which is something we expanded in this thesis. The paper was co-written by two of the supervisors of this thesis: Daniël A. A. Pelsmaeker and Danny M. Groenewegen. This thesis is essentially an expanded version of this paper, with more content, details and explanation, also addressing some of the concerns raised in the reviews. Nonetheless, we felt that it was appropriate to include the original, unaltered paper in this appendix.