Syntactic Code Search with Sequence-to-Tree Matching

Gabriel Matute
Wode Ni
Titus Barik
Alvin Cheung
Sarah Chasins

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-93
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-93.html

May 10, 2024
Syntactic Code Search with Sequence-to-Tree Matching

by Gabriel Matute

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination

Committee

[Signature]
Professor Sarah E. Chasins
Research Advisor
04/30/2024

[Signature]
Professor Alvin Cheung
Second Reader
04/30/2024
Syntactic Code Search with Sequence-to-Tree Matching
Supporting syntactic search with incomplete code fragments

GABRIEL MATUTE, University of California, Berkeley, USA
WODE NI, Carnegie Mellon University, USA
TITUS BARIK, Apple, USA
ALVIN CHEUNG, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

Lightweight syntactic analysis tools like Semgrep and Comby leverage the tree structure of code, making them more expressive than string and regex search. Unlike traditional language frameworks (e.g., ESLint) that analyze codebases via explicit syntax tree manipulations, these tools use query languages that closely resemble the source language. However, state-of-the-art matching techniques for these tools require queries to be complete and parsable snippets, which makes in-progress query specifications useless.

We propose a new search architecture that relies only on tokenizing (not parsing) a query. We introduce a novel language and matching algorithm to support tree-aware wildcards on this architecture by building on tree automata. We also present stsearch, a syntactic search tool leveraging our approach.

In contrast to past work, our approach supports syntactic search even for previously unparsable queries. We show empirically that stsearch can support all tokenizable queries, while still providing results comparable to Semgrep for existing queries. Our work offers evidence that lightweight syntactic code search can accept in-progress specifications, potentially improving support for interactive settings.

CCS Concepts: • Software and its engineering → Formal language definitions; Software maintenance tools; • Information systems → Query representation; • Theory of computation → Tree languages.

Additional Key Words and Phrases: Code Search, Syntactic Analysis, Tree Wildcards

ACM Reference Format:

1 INTRODUCTION
When a developer pastes a fragment of code into their IDE’s search box, why do they not start seeing matches right away? If their search uses string search, the answer is probably that the search query is too specific—too dependent on whitespace, on formatting choices. If their search uses a syntactic search tool, the answer is probably that their code fragment is not a parsable expression. Say a developer labors over their search query until they think it is complete, but they reach the end and it produces no matches. Is there a logical error in the query or are there simply no relevant results in the codebase? How can the programmer get more information to help them move towards the correct query? As in other programming domains, live feedback during query authoring holds...
the promise of giving users (i) early feedback about their queries and (ii) information they can use to refine their goal. Unfortunately, most of the query fragments en route to a programmer’s target query may not be parsable program fragments. If our code search tools can only offer feedback for complete, parsable states, we deny developers important early feedback.

Lightweight syntactic analysis tools—i.e., tools that use a domain-specific language (DSL) that resembles their target programming language to specify syntactic patterns—are used in a wide variety of domains. For example, Semgrep [40] is a security-focused static analysis tool that uses syntactic patterns to detect vulnerabilities, Comby [45] is a language-aware search and replace tool that has been used for large-scale refactoring, and TXL [5] is a structural analysis and transformation tool that has been used for program analysis and instrumentation. Language-specific examples, like Haskell’s Retri [37] and Go’s gofmt [11], are often used for programmatic code edits.

At their core, all these tools rely on syntactic search to accomplish their goal: given some lightweight pattern specification—a code fragment that may or may not use placeholders—they find all the matching positions in the source code. Traditionally, this matching is performed by comparing the syntax tree of the pattern specification against the syntax tree of the source code. Thus, syntactic analysis tools start by parsing the query into a tree and then rely on standard tree matching algorithms to search the parsed source files. Since the pattern specification needs to be parsed with this approach, syntactic analysis tools require the code fragment in the specification to be complete—that is, parsable into a syntax tree. In contrast, partial, often unparsable, queries are useful and well-supported in textual search tools such as find-and-replace. Thus, we identify the parsability constraint as a limitation of existing syntactic search tools.

To address this limitation, we observe that lightweight syntactic search queries are parsable code—and thus the partial queries that a programmer produces en route to a complete query are usually still tokenizable, even if they are not parsable. As with so many programming domains, the query author creates a tokenizable fragment as they craft a complete specification. As such, we present a new architecture (Section 2) that (i) only assumes queries are tokenizable, but not necessarily complete, and (ii) relies on minimal extensions to an existing lexer. We define a query language (Section 3) that accepts partial queries. Finally, to provide support for expression placeholders, we develop novel matching semantics (Section 4) defining sequence-to-tree matching.

We implement these techniques in a new tool, stsearch (Section 5). To evaluate our approach, we collected a benchmark suite (Section 6) of real-world search queries. We then evaluate (Section 7) our tool against Semgrep, a current state-of-the-art, commercial lightweight syntactic search tool. Finally, we discuss the tool’s limitations and future work (Section 8) and situate our approach within the related work (Section 9). This work contributes:

- A search query language for expressing syntactic search queries and formal semantics capable of accepting partial—but tokenizable—code fragments as queries.
- A matching algorithm, STMatch, that underlies our implementation, capable of matching a token sequence with wildcards against the syntax trees of source code.
- An open-source implementation, stsearch, of our techniques, and an evaluation showing that it supports not only parsable, but also tokenizable but non-parsable queries.

Our evaluation shows that for existing complete queries, stsearch is comparable to Semgrep: stsearch’s different semantics only excludes 4.95% of the results that Semgrep matches in our benchmark. Meanwhile, stsearch successfully accepts and processes all tokenizable partial queries, often providing results comparable to the complete queries with fewer tokens.
passport.authenticate( // Pattern error
passport.authenticate($NAME, // Pattern error
passport.authenticate($NAME, { // Pattern error
passport.authenticate($NAME, [...], keepSessionInfo: // Pattern error
passport.authenticate($NAME, [...], keepSessionInfo: true // Pattern error
passport.authenticate($NAME, [...], keepSessionInfo: true, ...) // Pattern error

Listing 1. Partial queries that result in a parse error and, therefore, produce no results in Semgrep.

Listing 2. Searching with regex, Semgrep, and stsearch for uses of passport.authenticate in a codebase. Notice that stsearch supports partial queries, so it uses fewer tokens than Semgrep for comparable results. Meanwhile regex struggles with false positives and negatives.

1.1 Motivating Example

Consider a developer using the authentication library passport [13] and trying to ensure that the authenticate function (signature below) is used securely in their codebase.

passport.authenticate(name[, option])

Reading the documentation [12], they discover that the function provides an option called keepSessionInfo; if keepSessionInfo is true, the application preserves information after a user logs into their account. By default, keepSessionInfo is false, since it makes applications vulnerable to session fixation attacks. To improve security, the developer wants to search their large existing codebase for uses of authenticate that use keepSessionInfo option at all.
**String Search.** The developer starts with a tool for performing string or regular expression search, like the standard command-line utility `grep` or the search box of their preferred code editor. Perhaps they start with the simple string search below.

```javascript
passport.authenticate
```

This simple string search finds most of the relevant `authenticate` uses pictured in Listing 2. Notice that spacing of any kind around the dot between `passport` and `authenticate` will prevent a match. For example, in Line 24 of Listing 2, a programmer has put a newline after `passport`. Thus the developer’s simple string query will accidentally fail to find this usage.

**Regular Expressions.** Next, the developer wants to filter the results to those that pass an explicit option parameter. Since the first function argument name likely varies throughout the codebases, they switch to regex and add a greedy wildcard `/.*/` to match the first argument.

```javascript
/passport\authenticate\((.*,\)
```

Regular expressions are notoriously hard to use [27]. For example, a wildcard `/./` will not match newlines by default in most engines, so many common uses, like in Line 7, can be hazardously overlooked. On the flip side, even simple cases for the first argument, like nested calls (Line 15) or comments (Line 21), can lead to a vast number of false positives. Finally, even in true matches, the character range selected is unlikely to match the relevant construct due to these same issues, rendering the results useless for programmatic changes.

**Lightweight Syntactic Analysis Tools.** Programming languages are not regular languages, so regular expressions are incapable of fully expressing them. Even if the developer painstakingly encodes more language-specific syntactic information into the query, like irrelevant white space and comment syntax, regular expressions can only express patterns in regular languages, while modern languages are at least context-free, e.g., relying on nested parenthesis.

At this point, the developer might switch to a more expressive tool. Alternatives abound, but a natural next step could be lightweight syntactic analysis tools. In contrast to heavyweight syntactic analysis tools, which users write programs that explicitly traverse and manipulate the program abstract syntax tree (AST), lightweight syntactic analysis tools accept queries that look similar to the programs being searched. For instance, our developer could use the lightweight syntactic analysis tool Semgrep with the following query.

```javascript
passport.authenticate($NAME, {..., keepSessionInfo: $VALUE, ...})
```

In contrast to our developer’s regular expression attempt, this query matches all intended cases, even Line 24, despite the formatting, comments, and nested expressions. Note that in Semgrep `$NAME` and `$VALUE` are interpreted as expression placeholders and `...` as zero-or-more items.

Lightweight syntactic analysis tools perform matching over the parse trees of a given file, which means that they are capable of supporting more expressive patterns. For example, they usually ensure placeholders respect matching delimiters and nested sub-expressions, making them capable of expressing patterns outside of regular languages. They can also leverage a substantial amount of information about the source language, like the precedence and associativity of operators.

**Syntactic Search for Non-Parsable Queries.** Nevertheless, current lightweight syntactic analysis tools have strict requirements on the input query. Since they need a tree structure to search over a codebase, they need to fully parse the query into a well-formed tree. For example, Semgrep uses a parser that requires that the query is a complete JavaScript (JS) statement or expression. Therefore, partial queries like the ones shown in Listing 1 would result in a parse error, preventing the search with no matches surfaced to the developer.
In contrast, our tool, stsearch, can provide results even for partial queries. In our example, while the developer is crafting the query with existing state-of-the-art tools, most of the intermediate, partial specifications are invalid and result in no useful feedback to complete the query. The developer can instead use stsearch, which introduces support for tokenizable queries, even if they are not parsable. The developer can write the query below, where $\_\_\_$ is similar to an expression placeholder and $\ldots\ldots$ is similar to a zero-or-more items placeholder.

```
passport.authenticate($_, {... keepSessionInfo
```

Our stsearch tool leverages the same insight used in syntax highlighting: many code fragments are *tokenizable but not parseable*. stsearch provides results for all tokenizable states en route to a complete query, providing feedback and context to the developer for those tokenizable fragments. Queries in our language (Section 3) are a sequence of tokens, and we even implement stsearch by reusing and extending an existing lexer to handle additional wildcards. Importantly, since our queries may not be parsable, we cannot use traditional tree matching techniques.

Instead, we introduce a novel sequence-to-tree matching semantics (Section 4). Our algorithm can take as input: (i) a token sequence and (ii) the concrete syntax tree (CST) from a source file; and select matching slices in the tree. Our approach matches concrete tokens to tokens in the tree, but ensures that wildcards match a complete subtrees. This novel strategy thus handles partial, but tokenizable queries while still leveraging the structure of the concrete syntax tree similar to existing state-of-the-art syntactic search tools.

## 2 SYSTEM OVERVIEW

In this section, we describe the system architecture of stsearch. In particular, we contrast the stsearch structure with the structure of prior lightweight syntactic tools.

Syntactic search tools take as input a query and a set of source code files. They produce as output a list of matches—i.e., source code files ranges that match the provided query. We use the term *lightweight* to refer specifically to those with query languages that resemble the syntax of the source language, typically by reusing the source language’s existing infrastructure.

### 2.1 Architecture of Traditional Systems

Previous systems for lightweight syntactic search (e.g., [40]) use the pipeline pictured in Fig. 1a to process both the search query and the source code. In particular, note that both the query and the source code are run through a lexer and a parser. Thus this approach requires parsing the query.

We briefly describe the two stages of traditional pipelines:

1. The tool conducts **Query Processing** with a modified parsing pipeline. Usually the source language is augmented with additional syntax for placeholders or other search constraints, so the tool typically extends the lexer and parser to support the new syntax. After processing the query, the pipeline outputs a tree pattern that resembles the code syntax tree.

2. Next the tool conducts **Tree Matching** to match the pattern against the syntax tree generated by parsing the source code. The trees usually share the same structure, since they come from similar parsers, so matching can be performed using standard matching techniques (e.g. [14]). Existing tools include many practical optimizations, e.g., building a search index.
Fig. 1. Comparison of stsearch architecture against traditional systems.

(a) Diagram of traditional lightweight syntactic search tool architectures. Traditional tools use a lexer and a parser to generate a tree pattern, then rely on standard tree-to-tree pattern matching.

(b) Diagram of stsearch architecture. It uses a lexer—but no parser—to generate a pattern token sequence, then relies on our novel STMatch algorithm to perform sequence-to-tree matching.

2.2 Architecture of stsearch

To handle partial, non-parsable queries, stsearch removes the parsing step from query processing. See the stsearch pipeline in Fig. 1b. As such, our inputs are extended to include all tokenizable queries, but we must provide a novel matching engine to support token sequences as the query. We can no longer rely on classical tree matching techniques.

(1) stsearch performs **Query Processing** using just a lexer. To support wildcards (see Section 3), we might still need to extend the lexer to support new syntax as in traditional lightweight syntactic search tools. However, we no longer need to update the parser to account for all language constructs that should allow for potentially ambiguous wildcards.

(2) **Sequence-to-Tree Matching** is our novel technique (described in Section 4) developed to support matching a token sequence against the syntax tree. Since the token sequence and the leaves of the tree are created by the same lexer, stsearch can match tokens to leaves, but our algorithm also supports tree-aware wildcards. Due to the heterogeneous types, many known search optimizations might not be directly applicable.
Pattern $p ::= k + (sequence) \quad Token \quad k ::= \_ (subtree\ wildcard) \quad | \quad \ldots (siblings\ wildcard) \quad | \quad k_{lang}$

Fig. 2. Syntax for stsearch. We introduce $\_\_$ and $\ldots$ tokens to represent placeholders in search queries, while the $k_{lang}$ token stands in for any token allowable in the lexical specification in the source language. Note that although we use EBNF notation for clarity, the language syntax is actually regular.

3 QUERY LANGUAGE

Since traditional lightweight syntactic query languages are defined as extensions to the source language grammar, they are only able to parse and interpret patterns that correspond to a complete grammatical production or construct, like an entire expression or statement. As such, partial specifications, potentially encountered when authoring a complete query, are usually not recognized in the language and therefore yield no results at all.

Instead, we notice that partial queries are still tokenizable. Actually, many syntax highlighting tools rely only on tokenizing precisely to support editing incomplete code fragments. Tokenization already encodes meaningful details about the language: dropping insignificant whitespace, splitting distinct syntactic elements (e.g., names and operators), etc. Meanwhile, it is usually a local process, making it more resilient to incomplete code fragments than full on parsing.

Our query syntax (Section 3.1) allows the reuse of existing lexers for the source language. Similar to the strategy of traditional systems, our tool can reuse the existing language infrastructure to process the query. In fact, many modern languages already have separate lexing and parsing infrastructure, making very efficient lexers easily available.

Consequently, our query semantics (Section 4.1) specifies results even for partial queries. Since we no longer produce a tree, we can no longer rely on standard tree matching algorithms to define matches for our language. However, we still want to be able to match against trees to preserve the expressivity improvements of syntactic search tools over regular expressions, e.g., to account for arbitrarily nested expressions. As such, we first outline the intuition of our language (Section 3.2), and then we give a formal specification of the matching algorithm in Section 4.

3.1 Syntax

stsearch accepts a code search query using the syntax shown in Fig. 2. As with many lightweight syntactic tools, a query is a string similar to a code fragment in the source language. In this case a pattern is a sequence of one or more tokens, where a token can be any token in the source language extended with the wildcards below. Note that, although we use extended Backus–Naur form (EBNF) notation for clarity, the language syntax is actually regular, as are the underlying tokens.

Our language supports two kinds of wildcards in a query:

subtree wildcard ($\_\_$) are similar to expression placeholders as found in most traditional syntactic analysis tools. They ensure that an entire subtree in the concrete syntax tree is matched, using the parse tree to properly express arbitrary nested expressions.

siblings wildcard ($\ldots$) are similar to zero-or-more items placeholders in many traditional syntactic analysis tools where they are used to match arbitrary sub-sequences in arguments, statements, or lists. They ensure that adjacent sibling subtrees are matched.
We now discuss the intuition behind \(\mathcal{B}\) without introspecting into the details of a specific \(\mathcal{F}\). For example, consider a query with just 3 wildcards (\(_\_\_\_\).)

Similarly, given the query on the bottom left, we want all trees that include a given property.

### 3.2 Intuition

We now discuss the intuition behind \(\text{stsearch}\)'s query language.

**Definition 3.2.1.** Let \(S^*\) be the set of finite sequences over some set \(S\), where \(s \cdot l\) (i.e., juxtaposition) denotes concatenation and \(s \mid l\) denotes that \(s\) is a sub-sequence of \(l\) for \(s, l \in S^*\).

**Definition 3.2.2.** Let \(T(\mathcal{F})\) be the set of finite ordered trees over some ranked alphabet \(\mathcal{F}\), where \(\mathcal{F}_p\) denotes the symbols in \(\mathcal{F}\) with \(p\)-arity, and \(\mathcal{F}_0\) is the set of leaves.

**Definition 3.2.3.** A concrete parse tree \(t \in T(\mathcal{F})\) is produced by a partial function \(\text{parse}(s)\) with \(s \in \mathcal{F}_0^*\) and the left inverse operation \(\text{yield}(t)\) defined by the in-order leaves of \(t\).

\[
\text{parse}(s) = t \implies \text{yield}(t) = s
\]

Given a query pattern \(p\) and a concrete parse tree \(t\), we want to define if there is a match, i.e., if it will be surfaced by our tool. Our goal is to ensure that a full, parsable query is guaranteed to match at least the same results as its parse tree. Meanwhile, a partial query should include the matches for the parse trees of all valid completions of the provided query to guarantee that the matches for the intended query results are included.

For a query with only concrete language tokens, i.e., without wildcards, it suffices to check if the pattern is a sub-sequence of the tree leaves, i.e., if \(p \mid \text{yield}(t)\). If the query is parsable, then \(p\) trivially matches \(\text{parse}(p)\), given that \(\text{yield}(t) = p\) and \(p \mid p\). Meanwhile, for partial queries, any parsable completion with a prefix \(l\) or a suffix \(r\) would also match.

\[
\exists l, r \text{ s.t. } \text{parse}(l \cdot p \cdot r) = t \implies p \mid \text{yield}(t)
\]

Once we consider queries with wildcards, defining a match becomes tricky. We want to use the parse tree structure to match more than regular languages, so we cannot rely only on sub-sequence matching. However, even with a full query there is no straightforward path to parsing wildcards without introspecting into the details of a specific \(\text{parse}\) function and choosing a resolution to any ambiguities. For example, consider a query with just 3 wildcards (\(_\_\_\_\_), it can either be parsed as two unary operators (matching \(-+\)s) or a binary operator (matching \(\times\times\)).

Instead, to match the intuitive behavior of traditional systems placeholders, we want to ensure that each wildcard matches entire subtrees (like a nested expression or statement). Meanwhile, for
concrete tokens, we want to keep the previous sub-sequence semantics to match any possible parse. In practice, this means we want match all possible valid parses given by replacing the wildcards with some complete syntactic structure, uncovering all possible parses for a given query.

Consequently, as shown in Fig. 3, we want every concrete token to appear in order in the tree. We want every *subtree wildcard* to match one subtree immediately after the last matched token. A *siblings wildcard* has a similar constraint, but it can match zero or more adjacent siblings.

### 4 SEQUENCE-TO-TREE MATCHING

Given a *pattern sequence (with wildcards)*, we first state the matching semantics as recognizing the *regular tree language* defined by the pattern (Section 4.1). Our intuitive notion, can be formalized by translating a pattern into a *tree automaton* that recognizes matching trees. We then present a novel STMatch algorithm that takes a pattern sequence and a tree cursor, i.e., a position in a tree, and checks directly for a match (Section 4.2). We outline the minimum requirements on the underlying interfaces and walk through the core algorithm components.

#### 4.1 Semantics

**Definition 4.1.1.** A *pattern* \( p \) is a sequence of tree leaves, potentially with wildcards \( \mathcal{W} \), i.e. \( p \in P^* \) where \( P = \mathcal{F} \cup \mathcal{W} \), where the wildcards \( \mathcal{W} \) might include the *subtree wildcard* \( _\$ \).

Given a tree \( t \), we want to check if it belongs in the "tree language" of a pattern \( p \), so we translate a pattern into a tree language specification. In particular, our intuitive notion outlined in Section 3.2 can be encoded in a recognizable tree language as defined by *finite tree automaton* (similar to languages over sequences or *word languages*). Therefore, we outline how to derive a *tree automaton* from each pattern to reduce the sequence-to-tree matching problem to *membership* checking.

**Definition 4.1.2.** A *pattern* \( p \) **matches** a tree \( t \in T(F) \) when \( t \) is *accepted* by the following *top-down nondeterministic finite tree automaton* [4, top-down NFTA]

\[
\mathcal{A}(p) = (Q, \mathcal{F}, I, \Delta)
\]

where

\[
Q = \{ s() \text{ such that } s \mid p \} \quad \text{(all states)}
\]

\[
I = \{ p() \} \text{ where } I \subseteq Q \quad \text{(initial states)}
\]

\[
\Delta = \left\{ \begin{array}{c}
q(f_0) \rightarrow f_0 \\
q(f(x_1, \ldots, x_n)) \rightarrow f(x_1, \ldots, x_n) \\
q(f(x_1, \ldots, x_n)) \rightarrow f(q_1(x_1), \ldots, q_n(x_n))
\end{array} \right\} \quad \text{(transitions)}
\]

Conceptually, a *top-down tree automaton* traverses a tree from the root to the leaves, associating a *state* with each *subtree*. It starts, by associating an *initial state* to the entire tree. Then, at each step, it propagates the state from the subtree root to its children, according to a set of transitions. Finally, the automaton accepts a tree if it is able to complete a traversal of the entire tree.

In our case, we want the states to track what part of the pattern each subtree matches. As such, we define *all states* to include any possible sub-sequence of the pattern \( p \). Furthermore, we want to ensure the full tree matches entire pattern, so the *initial states* only contain the full pattern \( p \). Finally, we specify the *transitions*, given a pattern state and a subtree:

- If the pattern state consists of a single leaf \( q = f_0 \) and the subtree is the same leaf \( f_0 \), then the pattern and the subtree match, so we finish the traversal of this branch.
If the pattern state consists of a wildcard \( q = \cdot \), then we can always match the entire current subtree \( f(x_1, \cdots, x_n) \), so we finish the traversal of this branch.

If the root \( f \) has \( n \) children \( x_1, \cdots, x_n \) we can then split the pattern state into \( n \) sub-sequences such that \( q = q_1 \cdots q_n \) and continue the traversal at each child.

Example 4.1.3. Let the alphabet \( \mathcal{F} = \{ \text{binop}(\cdot, \cdot), 1, + \} \) where \( \text{binop} \) might represent a binary operation (with two terms and an operation). Let the pattern sequence \( p = 1 + \cdot \cdot \cdot \cdot \) such that we have an automaton with the following transitions.

\[
c(f_0) \rightarrow f_0 \quad \text{when } c = f_0 \quad (1)
\]

\[
\cdot \cdot \cdot (f(x_1, \cdots, x_n)) \rightarrow f(x_1, \cdots, x_n) \quad \text{for all } f \in \mathcal{F} \quad (2)
\]

\[
q_1 q_2 q_3 (\text{binop}(x_1, x_2, x_3)) \rightarrow \text{binop}(q_1(x_1), q_2(x_2), q_3(x_3)) \quad (3)
\]

With this automaton, we can check the tree \( t = \text{binop}(1, +, 1) \) with the following trace:

\[
p(t) = 1 + \cdot \cdot \cdot \cdot (\text{binop}(1, +, \text{binop}(1, 1)))
\]

\[
\rightarrow \quad \text{binop}(1(1), +(+, \cdot \cdot \cdot \cdot )) \text{ by Rule 3}
\]

\[
\rightarrow \quad \text{binop}(1, +, \cdot \cdot \cdot \cdot ) \text{ by Rule 1 (twice)}
\]

\[
\rightarrow \quad \text{binop}(1, +, \cdot \cdot \cdot \cdot ) \text{ by Rule 2}
\]

Since we are able to traverse the entire tree we have that \( p \) matches \( t \), as we would expect.

Notice that without the second case, i.e., Rule 2 (for a \( \cdot \cdot \cdot \cdot \) ), the automaton simply checks the pattern corresponds to the leaves of the tree. This behavior matches our previous intuition for concrete patterns in Section 3.2, namely that \( p \mid \text{yield}(t) \). Therefore, the automaton presented is a generalization of those outlined semantics to account for subtree wildcards.

Extending the automaton to support more wildcards is straightforward. We can encode their semantics, including special structural constraints, by adding rules to the transitions. For example, for the sibling wildcard from Section 3.1, we would use the following.

\[
q(f(x_1, \cdots, x_n)) \rightarrow \text{binop}(q_1(x_1), \cdots, q_{j_1-1}(x_{j_1-1}+K_{i-1}), \cdot \cdot \cdot, G_{x_{j_i}+K_i-1}, \cdot \cdot \cdot, q_{j_i+1}(x_{j_i}+K_i), \cdots, q_m(x_n))
\]

with \( i \in [1, s] \), \( j_i \in [1, m] \) and \( K_i = \sum_{l=1}^{i} k_l \) for some \( k_l \geq 0 \)

when \( q = q_1 \cdots q_m \) where \( q_{j_i} = \cdot \cdot \cdot \cdot \) and \( n = m - s + K_s \)

Conceptually, given an \( m \)-split of the pattern state \( q \) with \( s \) sibling wildcards on each of the \( j_i \)-th sub-sequences, the rule continues the traversal at each child, similar to the last rule in the original transitions. The states \( q_i \) not corresponding to selected sibling wildcards are moved as-is to in-order child nodes \( x_j \) while the selected \( q_{j_i} \) states are replaced by \( k_i \) subtree wildcards. Consequently, each sibling wildcard matches \( k_i \) adjacent subtrees under the parent with root \( f \).

Similarly, although our automaton requires a pattern to match an entire tree, we can easily use our approach to match a slice of a tree, i.e., a new tree. For example, when searching for partial queries, intended matches are often part of a larger tree (as shown in Fig. 3), so we want more than just recognizing a match. Instead we consider all possible slices for a tree, where a tree slice is the range of all nodes between any two branches as a separate tree and find slices that match.
Table 1. Cursor interface used for the \textit{STMatch} algorithm.

<table>
<thead>
<tr>
<th>Pre-order Cursor overview</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next_subtree(self) \rightarrow Optional[Cursor]</td>
<td>Return a \textit{new} cursor to the \textit{next subtree} in pre-order after the \textit{self} node if it exists.</td>
</tr>
<tr>
<td>first_child(self) \rightarrow Optional[Cursor]</td>
<td>Return a \textit{new} cursor to the \textit{first child} of the \textit{self} node if it has any children.</td>
</tr>
<tr>
<td>first_leaf(self) \rightarrow Cursor</td>
<td>Return a \textit{new} cursor to the \textit{first leaf} of the \textit{self} node (itself if it has no children). \textit{Note: can be implemented from first_child}</td>
</tr>
<tr>
<td>token(self) \rightarrow Token</td>
<td>Only for \textit{leaf nodes}, return the \textit{token} located at the \textit{self} leaf.</td>
</tr>
</tbody>
</table>

4.2 Algorithm

We now present a deterministic algorithm that implements the tree automaton from Section 4.1 using a pre-order traversal, while matching concrete tokens directly to leaves and backtracking to resolve any ambiguity matching wildcards. The algorithm does not require any explicit tree slicing, since it traverses the tree using a cursor, and can be slightly modified to locate the end of the match, such that only potential starting locations need to be considered.

Our algorithm can use any sequence interface to iterate over the pattern. We only need \texttt{first} and \texttt{rest} operators to get the \texttt{first} element and the rest of the list, respectively. To simplify our presentation, we describe our algorithm in Python in Listing 3, where we use Python’s iterable unpacking (i.e., \texttt{first, rest = seq}) to access the relevant elements at each step. We also check if the sequence is empty using Python’s collection truthiness (e.g., \texttt{if seq}).

Our algorithm requires a pre-order cursor to traverse the tree. We outline the expected methods for such a cursor interface in Table 1. The first two methods, next_subtree and first_child, restrict the tree traversal to be in pre-order, but do not require a visit to every node. Meanwhile, first_leaf is a convenience function that skips down the left spine of the tree to the very first leaf, and token allows the algorithm to inspect and match the leaves to concrete tokens. Notice that all methods are also pure, they do not modify the cursor, but instead return a new cursor.

Concrete tokens are only required to define equality (a \texttt{== b}), specifically between a token in the pattern and in the leaf of a tree to check for a match. Meanwhile, the \textit{subtree wildcard} tokens just needs to be different from regular tokens, in our case an instance of the \texttt{Wildcard} class.

\textit{STM}atch (Listing 3) \textit{Outline}. Conceptually, the algorithm recursively matches each token in the pattern against the tree. If the next element in is a concrete token, then the algorithm must match the leftmost (i.e., next) leaf in the tree. Therefore, the algorithm, starting in Line 13, traverses to the first leaf under the cursor checks for a match and continues with the next subtree.

If the next element is a wildcard, then the algorithm must match a (i) complete subtree that (ii) includes the leftmost leaf and that (iii) allows for a match if any exists. Therefore, starting at Line 7, it guesses the subtree currently under cursor is a match and continues with the rest of the pattern. If at any point the matching fails, the algorithm backtracks and retries with next subtree rooted on the left spine (i.e., the first child) until it succeeds or runs out of candidates.
def match(pattern, cursor):
    if not pattern or not cursor:
        return not pattern and not cursor

tok, *rest = pattern  # unpack first & rest

if isinstance(tok, Wildcard):
    while not match(rest, cursor.next_subtree()):
        cursor = cursor.first_child()
    if not cursor: return False
    return True

cursor = cursor.first_leaf()
return tok == cursor.token() \
    and match(rest, cursor.next_subtree())

Listing 3. STMatch implemented in Python.

Fig. 4. Example execution trace for STMatch. On the left we have the recursive call tree, using numbered markers to represent cursors into a tree. On the right, we have two tree slices showing the algorithm state: first after a mismatch and then with the final match after successfully backtracking to a wildcard guess.

STMatch Example (Fig. 4). We demonstrate the algorithm with a trace of calls to match on a tokenized query and a cursor (mapped by a number) to tree slices (on the right) as shown.

(1) First call matches the first concrete token authenticate to the first_leaf, so it makes a recursive second call with the rest of the pattern and the next_subtree.

(2) Second call matches the next concrete token (_) to the first_leaf, except this time the cursor is already at a leaf node, so it makes a recursive third call.

(3) Third call needs to match a wildcard $\_\_$, so it will guess the corresponding subtree:
   (a) First, it tries matching the node under the cursor and it makes a recursive call with the last concrete token \( \_ \) but that call fails to match.
   (b) Next, it tries matching the first_child instead and it makes another recursive call with the last token \( \_ \) which eventually succeeds.

STMatch Complexity. Overall, the algorithm has a worst-case runtime complexity of \( O(k \cdot d^{h+1}) \), where \( k \) is the query length, \( h \) is the number of wildcards, and \( d \) is the maximum depth of the tree. Conceptually, for each of the \( k \) tokens in the query, the algorithm traverses up to \( d \) nodes and then for each wildcard it might backtrack up to \( d \) times for each node along a left spine.
rules:
- id: assigned-undefined
  languages:
  - javascript
  - typescript
  message: '"undefined" is not a reserved keyword in Javascript, so this is valid Javascript but highly confusing and likely to result in bugs.'
  pattern-either:
  - pattern: undefined = $X;
  - pattern: var undefined = $X;
  - pattern: let undefined = $X;
  - pattern: const undefined = $X;
  severity: WARNING
  metadata:
  category: best-practice
  technology:
  - javascript
  license: Commons Clause License Condition v1.0[LGPL-2.1-only]

Listing 4. Example of a Semgrep rule, which finds variables shadowing `undefined`. The underlying queries that would have been extracted for our benchmark are highlighted.

In practice we expect $k$, $h$, and $d$ to be fairly small, since expressions tend to be short and shallow. When processed by stsearch (see Section 5) our real-world benchmark (see Section 6) had queries with a median length of 8 (max 31) tokens and a median of 2 (max 10) wildcards, while the corpus syntax trees had a median depth of 15 (max 907) nodes. Our performance evaluation (Section 7.3) also found that for these real-world uses the backtracking complexity was not an issue.

5 IMPLEMENTATION
To implement sequence-to-tree matching, we created a free-standing Rust implementation of the algorithm (Listing 3) using traits for the sequence and cursor abstractions described in Section 4.2. The STMatch algorithm together with the interface declarations is 76 lines of Rust.

To implement our source code parser (Section 2.2), we used the Tree-Sitter [3] Rust bindings and tree-sitter-javascript, to generate an efficient, flexible JavaScript (JS) parser. Our syntactic search implementation wraps the concrete syntax tree produced by the parser, to implement the cursor interface (see Table 1) required for the presented STMatch algorithm.

Since Tree-Sitter provides error-tolerant parsing, we reuse the source code parser to generate the query tokens by ignoring any parse errors and extract the leaf tokens. By leveraging a query language with a compatible syntax (see Section 3.1), stsearch contains only 7 lines specific to JS. stsearch is open-source and publicly available at plait-lab/stsearch.

6 BENCHMARK SUITE
We created a benchmark suite of queries (Section 6.1) from the existing Semgrep [40] ecosystem and collected a corpus of source code (Section 6.2) from the npm [35].

6.1 Query Collection
Semgrep is a static analysis tool for finding bugs and vulnerabilities in source code. As such, they have a standard repository semgrep-rules [41] of analyses covering many languages and
frameworks. Each rule (e.g., Listing 4) contains complete queries joined by conjunctions and disjunctions, as well as other operators to specify the relative placement of matches.

Overall, we extracted 308 unique queries for a popular library: the Express [8] framework. For each of the 52 Semgrep rules for Express, we extracted and canonicalized each query by normalizing white space, anonymizing all placeholders, and removing syntactic sugar.

On stsearch translation. Our tool has slightly different syntax than Semgrep, so we must translate each query. First, Semgrep placeholders start with a $ followed by an uppercase name, while for simplicity stsearch only supports anonymous wildcards $_. Second, Semgrep needs separators (e.g., commas for lists) when using a zero-or-more placeholder, but stsearch does not assume token semantics and interprets them literally, expecting a corresponding node in the tree. Our translator converted Semgrep queries into the equivalent stsearch queries.

On tokenizable prefixes. Finally, we computed 1107 unique unambiguous partial tokenizable queries from these complete queries. To generate unique partial queries, we tokenized each complete query with Pygments [2], a standard Python tokenizer, then took ranges of token prefixes to construct canonicalized and, consequently, unique and unambiguous partial queries.

6.2 Corpus Collection
To create a corpus on which to run our suite of queries, we sampled a corpus of 100 repositories of npm packages. To make sure they were relevant, we selected packages that directly depend on Express and do not list typescript as a required dependency, since stsearch currently only supports JavaScript (JS). Because npm is a package registry and some packages do not publish their source code, we also required that they listed a public GitHub repository with their source.

Overall, the corpus contains 15 233 files. The average size is (10 ± 190) kB (mean ± std. dev.) with 99% of the files under 130 kB, but the maximum size at 5.1 MB. After inspecting a sample of the large files, it seems that the unusually large files are the result of automatically generated outputs committed to the repositories. Given that these files are included in source repositories, we include them in our analysis, but they are unlikely to be relevant to developer queries.

7 EMPIRICAL EVALUATION RESULTS
We evaluate stsearch using our benchmark queries on our benchmark repositories (Section 6), using Semgrep [40] as a baseline. Overall, we aim for our tool to offer results for partial queries, while remaining comparable to existing tools for complete, parsable queries. Thus our evaluation centers on the following research questions, operationalized and investigated below.

**RQ1** How does stsearch semantics compare to established tools for complete queries?
**RQ2** How do stsearch results for partial queries evolve as tokens are added?
**RQ3** Can stsearch provide results at interactive speeds in practice?

7.1 Complete Queries
For RQ1, we compare the semantics of stsearch by inspecting the discrepancy in the results with respect to Semgrep on complete queries in our benchmark. We call a result excluded if a particular region of a particular source file is returned by Semgrep but not by stsearch. Conversely, a result is included if a particular region of a particular source file is returned by stsearch but not by Semgrep. We deliberately avoid using terms such as false positive or false negative because Semgrep’s results are not ground truth, simply a different attempt at delimiting relevant results.

---

1we excluded the polyfill-service repo, since it has 7x more files than the rest combined and would skew our results.
Table 2. Unique matches produced by running stsearch and Semgrep on all complete queries.

<table>
<thead>
<tr>
<th></th>
<th>Semgrep</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Matches</td>
<td>Excludes</td>
<td>Total</td>
</tr>
<tr>
<td>stsearch</td>
<td>6 025 658</td>
<td>(67.65 %) 12 603 448</td>
<td>18 629 106</td>
</tr>
<tr>
<td></td>
<td>(4.95 %) 313 836</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>6 339 494</td>
<td>-</td>
<td>18 315 270</td>
</tr>
</tbody>
</table>

Fig. 5. Match disagreements per complete parsable query between stsearch and Semgrep. The charts shows the breadth of total matches for each query and the distribution of query disagreements. The query exclusion / inclusion rate does not appear to be correlated with the quantity of results.

Since the current version of stsearch uses the input syntax tree as-is, we did not use Semgrep’s toggleable syntax tree rewriting passes. For example, Semgrep offers optional constant propagation as well as matching modulo associativity and commutativity of standard operators. Note that future versions of stsearch could also be extended to add semantics-aware features (see Section 8).

For some queries, both tools produced no results. Some analyses in the semgrep-rules [41] apply extremely rarely, so no relevant code snippets appeared in our corpus. Since these queries offered no information about the behavior of either tool, we dropped these queries. Furthermore, Semgrep was unable to correctly process 356 files due to internal errors. Thus, our discussion only details results for the 162 queries that produced matches and files with no errors.

We aggregate the results for complete queries in Table 2. The differences per query are in Fig. 5. We aggregate matches across tools by checking if the character ranges of the program are identical, i.e., if they start at the same character and end at the same character.

7.1.1 Exclusions. For 30.25 % of queries, stsearch included all results produced by Semgrep. Overall, across the benchmark suite, stsearch excluded 4.95 % of the matches identified by Semgrep. Semgrep leverages a semantic understanding of JavaScript (JS), while stsearch currently operates over the unaltered input CST using purely our language-agnostic approach. Below we include a brief description of a few resulting categories of exclusions.
**Semgrep leverages knowledge of source language semantics, e.g.**

- In Semgrep, the query `'express'` matches "express", since in JS there is no semantic difference between them. However, stsearch expects a literal match of every token, so a single quote ' will not match a double quote ".
- Semgrep ignores trailing commas and semicolons when matching, so the code fragments `[a,]` and `[a]` would match each other, but stsearch requires a literal match.
- Semgrep disregards the order of the keys in an object literal, but stsearch requires code snippets to match the order specified in the query. Note that JS defines the evaluation order inside object literals, so in this case stsearch is simply more conservative when matching, e.g., when matching `{a: f(), b: g()}`, Semgrep will also match a reordering of the keys, even if it might change the semantics of the program due to side-effects.

**Semgrep special handling of imports, e.g.**

- Given the query `$_VM.run(...)`, Semgrep will surface the snippet below as a result, despite it not having a member call expression. This behavior is not toggleable.

```javascript
const {run} = require('sandbox');
run('1 + 1', (res) => console.log(res));
```

**7.1.2 Inclusions.** stsearch produces more results than Semgrep, generating 67.65% additional matches for the benchmark suite. Recall that stsearch operates as though every query may be partial, and thus offers partial matches even for these parsable queries. For example, if we write a query to match assignments: given the query `$_ = require('express')`, stsearch would produce a partial match for the code below, identifying the highlighted match below.

```javascript
const express = require('express');
```

Since Semgrep must match an entire tree, and since this line of code both declares and assigns to `express`, Semgrep does not include this match. With the `vardef_assign` setting on, Semgrep could match the entire declaration. No setting would allow Semgrep’s result to exactly match stsearch’s (yellow-highlighted) match range with a single query.

**7.2 Partial Queries**

To answer RQ2, we measured how many results are filtered by each token prefix for each complete query and how that process converges to the final set of results. Throughout Fig. 6, each row represents one completed benchmark query, and each cell in the row represents an intermediate, tokenizable query en route to the complete query, with a token added per column. Note that Fig. 6 also includes a distribution of token lengths for complete queries in our benchmark suite.

Recall that by construction stsearch ensures that a tokenizable query results always includes all matches for any potential token completion (see Section 3.2). Thus, the results for each intermediate query necessarily include all results associated with the corresponding final, complete query. Therefore our main questions here are: (i) what is the impact of each token, and (ii) how many additional results stsearch includes, beyond those for the complete query.

Overall, in Fig. 6a the first few concrete tokens (the first is usually a wildcard) do the filtering, while in Fig. 6b most queries converge on the final results long before the last token. An interesting exception occurs for a group of queries with 6 tokens; starting with `$_ [require] [express]` that search for specific library imports. We see these tokens are effective at filtering matches; however, basically all imports contain this prefix. Therefore, they do not converge on the final results until the specific library is included in the query (e.g. `express`), but the last `]` is then redundant.
Fig. 6. **stsearch results progression for each token prefix en route to a complete benchmark query.**

We investigate how results are filtered by each additional token and how they converge towards the final set. Recall that by construction (Section 3.2) adding a token can only result in a subset of the previous matches.

For both charts, each row corresponds to complete benchmark query, while each cell represents the hypothetical partial query resulting from the $n$-token prefix of the corresponding complete query.

(a) **Selectivity of each additional token.** We graph the results filtered by each new token (in violet) to identify key tokens for each of the complete queries. For the first column (since there are no previous results) we use the query with a single sibling wildcard (i.e. the one with most matches) as the set of previous results. Notice that the first few concrete tokens (the first is usually a wildcard) do most of the match filtering.

(b) **Convergence into completed query.** We graph the in-progress results ultimately included (in green) in the final results for the completed query (in blue), i.e., the precision of results for a query prefix search. Notice that the results often converge towards the results for the full query before the last token.
Table 3. stsearch execution time per file, for all files in our code dataset and all queries in our query dataset. We separate the parsing time, since the former should only have to be performed once per file.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Average ± SD</th>
<th>Median</th>
<th>99th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree-Sitter [3] parsing</td>
<td>(3 ± 31) ms</td>
<td>660 µs</td>
<td>29 ms</td>
</tr>
<tr>
<td>stsearch searching</td>
<td>(10 ± 480) ms</td>
<td>270 µs</td>
<td>24 ms</td>
</tr>
</tbody>
</table>

7.3 Performance

To answer RQ3, we measured stsearch’s execution time for the 308 complete and 1107 partial queries in our query suite, on each of the 15 233 files in our code suite. We used a server with an Intel Xeon CPU E5-1680 v2 and report the parsing and searching execution times in Table 3. Notice that for 99% of searches, stsearch takes less than 24 ms to find all matches, while the maximum search time was 230 s for a large, automatically generated file (see Section 6.2).

We conclude our non-optimized prototype is already performant enough to provide live feedback at interactive speeds. Assuming we have parse trees of all the files in a repository, we could complete a search in under one second for 91.10 % of the repos in our benchmark. Note that this assumes a naive single-threaded approach, searching each file in sequence rather than in parallel. In addition to being trivially parallelizable, we anticipate many other opportunities for effective optimizations, e.g., via a search index or incrementalizing results.

8 DISCUSSION

We now discuss the practical benefits and limitations of our approach. We also propose interesting directions for future work on stsearch.

Supported Languages. Our approach can support any language for which we can generate a syntactic tree, including all deterministic context-free languages. Implementing our technique does not require modifications to the grammar or parser implementation; so (i) the language and parser can evolve without requiring modifications to our tool and (ii) we can support new languages in stsearch without engineering custom parsers. In contrast, previous systems (see Section 2.1) must modify both the grammar and the parser to account for placeholders.

Furthermore, we expect error-tolerant parsers, capable of producing meaningful trees in the presence of syntax errors, to enable our approach to support in-progress codebases. Our benchmark already includes files that Semgrep [40] was unable to parse (and therefore search), while stsearch was able to process every file using the standard error recovery in Tree-Sitter [3]. The specific error handling strategy will have an impact on the matches for ill-formed code; e.g., a panic strategy that discards tokens might unintentionally exclude matches.

Grammar and Usability. Although our technique does not require modifications to a language’s parser, the behaviors of all lightweight syntactic approaches are ultimately affected by grammar and parser design. In particular, two grammars for the same language may group tokens differently. For example, to avoid left recursion, a parser might parse an infix operation like a+b as the tree \( \text{infix(a, op(+, b))} \), such that our subtree wildcard could unexpectedly match +b.

For stsearch, we used a Tree-Sitter grammar, which aims to have an “intuitive structure.” Semgrep uses the same grammar as the starting point for its custom grammar, so we matched the matching behavior of Semgrep for our evaluation. A different grammar will affect the matches for a given lightweight syntactic search tool results, potentially diverging from user expectations. Future work should explore the usability of lightweight syntactic tools.
Supporting Semantic Analyses. Although st:search currently does not use language semantics, our technique can be extended to leverage semantics knowledge by manipulating the search tree rather than the query. This allows us to maintain the core insight of our approach, i.e., only tokenizing potentially incomplete queries to search over complete, parsable source code.

Given that our matching semantics (see Section 4) are defined over trees, we can support many analyses that can be encoded as tree modifications. Consider the examples in Section 7.1: tokens with equivalent semantics (e.g., `'` and `''`) can be canonicalized before matching; insignificant tokens (e.g., trailing `,`) can be dropped from the tree so they are disregarded. More complex analyses (e.g., constant propagation) could be supported by matching the query against a tree encoding the transformed source program (e.g., replacing a subtree with an inferred constant).

Going further, one could perform matching modulo associativity and commutativity, by considering all possible trees for an expression, or match type information by using a type-annotated tree. We expect the complexity and performance costs of these approaches to vary wildly, and some may be irreconcilable with the goal of maintaining interactive speeds. Echoing the discussion above, we expect future research may need to assess the need for and usability of such features.

9 RELATED WORK
Prior work has studied developers’ code search strategies [39] and existing techniques [25] to support them. Our approach provides an alternative to traditional tree pattern matching techniques by leveraging prior work on tree languages. We extend this work to create lightweight tools for program analysis and source-to-source transformations.

9.1 Program Analysis and Transformation
Lightweight Syntactic Tools. Existing tools leverage lightweight specifications for analysis and transformations. They aim to hide AST details behind a declarative syntax that leverages the source language (see Section 2.1). Throughout this paper we compare against Semgrep, but TXL [5] and Comby [45] (which also powers [43]) also have a lightweight query syntax and include support for multiple languages. More narrowly scoped tools exist, with Cocci:nelle [24] as a notable mention for its successful deployment for API evolutions in Linux [23].

However, every one of these tools require that the input query be parsable as a tree structure. We contribute a reusable technique to handle partial queries to the existing techniques.

Heavyweight Language Frameworks. Many languages have frameworks to analyze and transform source code programmatically. For example, for JavaScript (JS) the extensible ESLint [7] linter, jscodeshift [18] “codemod” toolkit, and the recast [33] library provide direct access to parse, analyze, and manipulate the AST for a JavaScript program. These frameworks tend to be more powerful than their lightweight syntactic counter parts, since they can express arbitrary constraints.

Cubix [19] (introduced by [20]) even extends this approach to support multiple languages with a single query. Recently, YOGO [36] was built using this framework, such that it is capable of performing a semantic search over multiple languages. Other work like [30], which uses island grammars [29], aims instead to be easily extensible to new and ad-hoc languages.

There are many tools whose focus is to collect and query source code information, like [21] and [10]. Some have an increased focus on their query language, like CodeQL [6] and [44]. There are even tools that rely only on tokenizing the source code to avoid parse errors, like Cobra [16].

However, these tools require significantly longer specifications, which often include large amounts of boilerplate. Furthermore, their DSLs are usually embedded in languages without any support for partial programs or even program sketches.
API Exploration. Another interesting direction explored in the literature is the search needs specific for API exploration. For example, Strathcona [15] automatically assists developers to find relevant examples, SSI [1] supports inspecting entities based on their API usages. Meanwhile, Examplore [9] provides an interactive interface to learn APIs through existing usages.

9.2 Tree Search and Matching

Regular Tree Languages. Regular trees and their properties have been studied in the prior literature. We leverage existing work (see [4]) to describe and characterize our technique in Section 4.1.

Similar to regular languages, each tree automaton recognizes a tree language that can also be encoded by a regular tree expression. Therefore, queries for stsearch can also be directly expressed using a regex-like notation designed for tree languages. However, this notation must also encode a tree, such that the query must still be parsed and cannot be incomplete.

Tree Pattern Matching. Searching and matching a tree pattern in a larger tree is a common problem in a variety of domains, including automated reasoning, compiler optimizations, and syntactic search. Although technically it constitutes a subset of the general regular tree expression matching problem, it has been separately studied and optimized [14]. However, as described earlier, solutions to this problem presume we can parse a tree from a query specification.

Tree Query Languages. Many tools exist that provide a query language to search over tree-like structures. For example [32, 31] and [22] provide a DSL to search over syntax trees. Meanwhile, the Rosie Pattern Language [17] aims to be a reusable pattern language more powerful than regex. However, the languages differ from their target, so they are not as lightweight.

9.3 Program Transformations Synthesis

Identifying Edit Locations. Several tools have explored automatically synthesizing program transformations. For example, LASE [26] and Refazer [38] are able to generalize from examples to automatically produce an edit script. In general, the synthesized program must include a way to identify the relevant locations to edit or a syntactic match specification.

By construction, these tools produce trees to specify the edit locations and even the rewrites, since partial specifications were not supported. We hope our work opens the opportunity to operate and surface partial specifications as targets for synthesis.

Interactive Transformations. A variety of interactive tools have leveraged program synthesis to deal with the challenges of authoring program transformations specifications. In particular, Blue-Pencil [28] and Overwatch [46] leverage the interactive history to automatically suggest rewrites to the developer. Meanwhile, reCode [34] uses the find and replace interaction for specification. On the other hand, ALICE [42] focuses on search through an interactive specification.

However, these tools have the same limitations as the underlying synthesis engines and are unable to produce or operate on incomplete queries. Therefore, we hope that supporting partial queries is a step in and of itself to make program transformations more accessible.

10 CONCLUSION

In this paper, we introduced a new architecture to support lightweight syntactic search with partial, but tokenizable queries. We formalize a query language and present stsearch, an implementation of these techniques evaluated on a real-world benchmark. We found that our approach can effectively support in-progress queries, while providing state-of-the-art results for completed queries.
ACKNOWLEDGMENTS

The authors would like to thank Federico Mora Rocha and Justin Lubin for their suggestions and discussions regarding the theoretical contributions in the paper.

This work is supported by NSF grants FW-HTF 2129008, and CA-HDR 2033558 as well as by gifts from Google and EPIC Lab sponsors G-Research, Adobe, Google, Microsoft, and Sigma Computing. Sarah E. Chasins is a Chan Zuckerberg Biohub Investigator.

REFERENCES


[40] Semgrep. Version 1.15. r2c, Mar. 15, 2023. URL: https://semgrep.dev.


[43] Source Graph: Batch Changes. URL: https://docs.sourcegraph.com/batch_changes.

