RFC 75 REVIEW: The Search Query Language: Input Representation and Processing

Editor: rijnard@sourcegraph.com
Date: November 2019
Status: REVIEW
Required Approvals: Nick, Christina (Please review by Dec 21)
Approvals: Tomás, Nick (there isn't really much to approve here other than to move forward with an actual proposal for a language for our search syntax), Stephen, Joe

Background and motivation

There is currently a mix of concerns for representing and processing search inputs resulting in ad-hoc handling and related bugs [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. These issues will persist (and we will continue to pay tech debt) until there is an intentional effort to handle the concerns that these changes entail. It also creates a barrier for introducing new features, syntax, runtime behaviors, and defaults.

Our search query input is a language. Over time this language has evolved, and effort has been directed at achieving functional behavior, and thus the source of truth for what our language means and does is foremostly codified in the backend implementation (we do have, e.g., documentation on its subparts, but how these interact is sometimes undefined, and is not a complete technical description of its properties). We lack a concentrated effort that defines what our language comprises and how its subparts interact. This proposal recognizes the opportunity to resolve these concerns, and calls for a conscious decision in design tradeoffs around the concerns and issues in our search language as enumerated in the rest of this document. The hope is that some classes of recurring issues can be handled "once and for all", and that a reevaluation of the language can inform new designs and features. The aim of this document is (a) to make the (known) concerns of search inputs and handling explicit, and (b) to outline a solution that enforces a clean separation of these concerns. We are in the fortunate position where we can be structured about the design of a Code Search Language based on our current experiences and knowledge in a way that is far ahead of other approaches. Let's seize it.

There has been a lot of historic discussion on search, and changes to the GQL interface, and so on, and there are known issues. This RFC does not attempt to directly suggest concrete changes that import the complexity of GQL or UI/UX issues, but rather to inform their design and changes by what we have learned from the current approach to search input, the evolution of the language around it, and steps toward defining what a search query is.

Thus: this RFC does **not** seek to directly propose:

- a UI or UX decision around search
- a concrete change to the GQL interface

Instead, this document proposes a **search model** as an initial step that addresses concerns in (a) accommodating the varieties of search input representations and processing now and going forward and (b) any consumer of the input (whether a client webapp, a UI/UX component, or backend search process) can reference a well-defined specification of what the input comprises.

It is also challenging to refer to search input and its parts (in code and conversation, we use “query”, “pattern”, “fields”, and so on, sometimes in ambiguous ways). This document seeks to establish a set of terms for referring to search inputs. Please refer to the **Glossary** as needed.

The desired outcome for this RFC (the thing being approved) is a consistent viewpoint and agreement on how to treat search inputs from a language perspective, and a commitment to consider the language design explicitly in decisions around changes to the API, user interface, or backend implementation. This foundation hopes to inform how we approach search input from an implementation perspective, what to consider when implementing changes from a product perspective, and to help obviate confusion and ambiguity around search processing and behavior.

Short version

You can navigate this document based on what seems particularly relevant or interesting to you. The problem section first describes the current search input representation and search behavior. The second part explains what the goals and concerns are for search functionality, and how they relate to challenges in input representation and processing. Four concrete challenges are identified. The proposal section explains a search model for overcoming these challenges, describes the advantages, and proposes next steps.

Problem

Outline

To communicate the problem, this section first deconstructs the state for how we currently treat inputs. It then discusses how we process this input representation to direct search functionality.

The last subsection describes a conceptual representation of search queries, how it relates to our goals for search functionality, and the challenges we face in separating concerns.

The current state: A deconstruction of how we represent and process search input

The intent of this section is to simply describe “what is”; no judgment or proposal is made based on the current state. We start with a concrete walkthrough of a search query for teasing out concerns. Bolded terms are entries in the **Glossary**.

Our user input for a **search query** is a single, contiguous string. A typical search query that uses **search keywords** (reference) might look like this:

```
repo:^github\.com/sourcegraph/sourcegraph$ file:.go case:yes TODO
```

Taking this as an example for our default literal search behavior (and putting aside for the moment other **search forms**, e.g., diff and commit search) here are the notable properties of information that the search query encodes, and the assumptions we make about it:

1. The `TODO` string above is our **search pattern**, and expresses what content we want to match inside a file. It can be any string. It is special because it is not prefixed by a **search keyword**.

2. We assume whitespace separates independent **terms** in the query: whether it is a **search keyword** (e.g., `file`) with an associated **value** (e.g., `.go`) or a **search pattern.**

3. We *infer* from the string in the query `TODO` is the search pattern because we can detect known **search keyword**s, as separated by *whitespace* thanks to our assumption in (2), and then accept the search pattern `TODO` because a prefixed search keyword is absent from it.

4. Due to the logic in (3), the order of search keywords and search pattern is not significant. We can apply that same logic regardless of where `TODO` occurs in the **search query**. Thus, the following inputs are considered equivalent:

```
repo:^github\.com/sourcegraph/sourcegraph$ TODO file:.go case:yes
```

```
TODO repo:^github\.com/sourcegraph/sourcegraph$ file:.go case:yes
```

5. We place constraints on the **value**s that **search keywords** can take on. These values may be restricted to a set of variants (or enums). For example, `yes` and `no` values exist for the `case` search keyword. Alternatively, a valid regular expression can be used for the `repo` search keyword. We enforce the kind of **value** that a **search pattern** may have with **types**.
6. Some **search keywords** are special and affect the *interpretation* of the **search pattern**. The `case` keyword is one such special keyword: it will force our **search pattern** to be matched case sensitively.

Not shown in the example, additional significant properties of search queries are that:

7. Some keywords may be specified more than once.
8. Some keywords are incompatible with others.
9. Some keywords may be negated.
10. Some keywords have default values.
11. The **search pattern** may be omitted. This can trigger other **forms of search** (e.g., repo search).
12. Search keyword values may be enclosed in single or double quotes (e.g., we may write `case:'yes'` or `case:"yes"`). Such quotes *do not* change the *interpretation* of the **values** for the **search keywords**.

The properties above are, for the most part, consistent for a particular **search kind** (or **search option**), which may be one of `literal`, `regex`, or `structural` at the time of writing. The backend definitions for some of the behavior and attributes above lives in searchquery.go.

The above query assumes that a `literal` search kind is in effect. Lifting that assumption, other search kinds can be in effect for a particular search query. The idea of a **search kind** warrants special attention because:

a. There are additional properties of the search query that are *specific to* the active **search kind**.
b. There are also additional properties of the search query that can *affect* which **search kind** is active (e.g., the `patterntype` **search parameter**).

Here are the notable properties of information that the search query encodes, and the assumptions we make about it, continued from the list above, but *with respect to* **search kind**:

13. As per (a) above, and in contrast to (12), the *interpretation* of the **search pattern** *may* be affected by quotation. It depends on which **search kind** is active. These properties are enumerated below.
14. As per (b) above, the `patterntype` **search keyword** can set which **search kind** is active. If `patterntype` is absent, we set a default based on configurations that are external to the **search query**.

For the `literal` **search kind**, by default:

15. Whitespace in the **search pattern** is significant. Searching for `to<space>do` is not equivalent to `to<space><space>do`.
16. The search pattern is interpreted case sensitively.
17. Single and double quotes in the **search pattern** are significant (i.e., we search for single or double quotes respectively in the file content).

For the `regexp` kind, by default:

18. Whitespace in the **search pattern** is converted to `.*?` (i.e., lazy repetition of any character), cf. 15. `to<space>do` will match the string `totalBytes?: double.`
19. The search pattern is interpreted case *in*sensitively (cf. 16). `to<space>do` will also match `totalBytes?: Double`
20. When the search pattern is single or double quoted, both (18) and (19) are not in effect, and the literal search kind is in effect (cf. 13). That is, in contrast to **search keyword value**s (cf. 12), the search pattern value can have one of two types for the regexp search kind: a quoted string (considered its own literal **type**) or an unquoted string (i.e., a regex **type**).
21. When the search pattern is delimited by `/`, as in `/to do/`, the slashes are ignored and the **search pattern**, i.e., `to do`, is interpreted as a regular expression (no conversion is done on the whitespace in the pattern, in contrast to (18)).

For the `structural` kind, by default:

22. Whitespace in the **search pattern** is treated insignificantly.
23. The search pattern is interpreted case sensitively.
24. The search pattern does not need to be quoted, and quoting has no effect. However, the search pattern usually *needs* to be quoted because the prevalence of colon syntax like “foo:[bar]” will conflict with parsing **search parameter** syntax like `foo:bar` and raise an error.

There are more behaviors here not listed explicitly: What constitutes valid and invalid queries (i.e., the interactions of **search patterns** and **search keywords** like `case`), and the suggestions, warnings, or errors that we show for such queries.

Processing a search query: A breakdown

It's useful to understand the processes that enable the above behaviors. I.e., how to achieve and enforce the desired search functionality, avoid bugs, and make our input language and behavior extensible.

The premise is that an input search query is part of a *language*. Which parts of our language a consumer (webapp client, backend, UI component) cares about, is up to the discretion of the implementation of such a consumer. A consumer need *not* implement a parser or type checker for the entire language, but it is important that we define a specification for our input language *should such a consumer want to* parse, inspect values, validate, or change the input in a well-defined way.

The process of converting an input **search query** string into a data structure for performing search consists of defining what we are allowed (and usually more importantly, *not* allowed) to express in our language. In general there are different layers of expression, each entailing different **checks**. The primary layers, or phases, that bear on converting search queries to a data structure for ultimately running a search are as follows:

A. *Syntax validation*

   We parse the string into an initial data structure, which we'll call a **parse tree**. The parse tree preserves all syntax elements (consisting of, e.g., tokens like strings, colons, and so on). The parsing phase:
   - understands lexical concerns how to deal with quoted strings and escape characters in such strings, but not what those quoted strings *mean* to us yet
   - groups related **terms** (like tokens that are part of a **search parameter**)
   - fails if a search query is syntactically malformed, according to what we consider to be correct or wrong syntax

B. *Semantic validation*

   This validation phase:
   - removes syntax we may no longer care about, where such a data structure is **valid** with respect to semantic criteria we care about. For example, a valid search pattern for the `literal` search kind, such as `*`, may not be valid for another search kind, like `regex`, and such a property must be checked.
   - checks whether sequences of tokens or groups of **terms** in the **parse tree** are legal, e.g., whether we allow a **search keyword** in our language.
   - produces a valid data structure (as above) which is our **abstract syntax tree**. The abstract syntax tree does not contain full information about the syntax of the query (this information is thrown away).

   Semantic validation isolates these sorts of checks after parsing, and with respect to *meaning* (i.e., are we checking whether a search query is valid for a literal search, or a

regex search?). Such checks only happen after we obtained a parse tree, i.e., after the point where we no longer care about the specific syntax of the search query.

C. *Query transformation*

We might want to change parts of the **search query** for various reasons. For example, we currently remove or combine **values** for **search keywords** that may be specified more than once or may normalize all the **search keywords** to be lowercase. Whether we do so before or after (A) or (B) depends on the information needed to perform the transformation.

These parts and their interaction are roughly illustrated by this diagram:

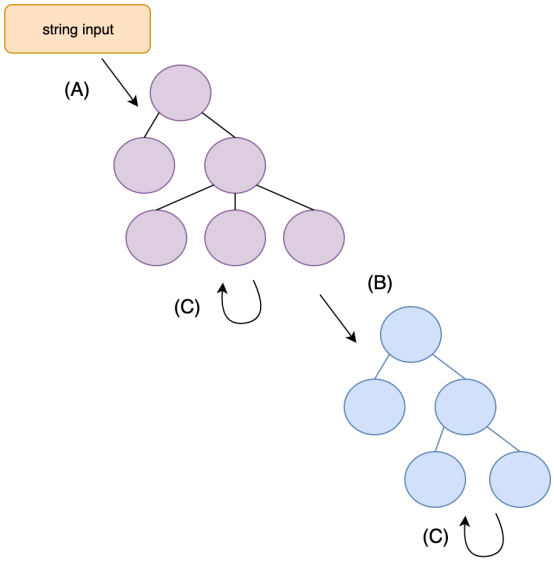

Fig. 1

Principles for separating search concerns and current challenges

Search representation overview

For discussion, we will again focus on file content search, and put aside other **search forms**, without loss of generality.

We can conceptually treat a search query's final data structure as a simple **abstract syntax tree**. One node (or **term**) being the **search pattern** (possibly empty for other **search forms)**, and the other node (or **term**) being an unordered collection of **search keywords** and their

values. We define a **search parameter** to be a term that consists of a single **search keyword** and its **value**. At a high level, the final representation of a search query looks like this:

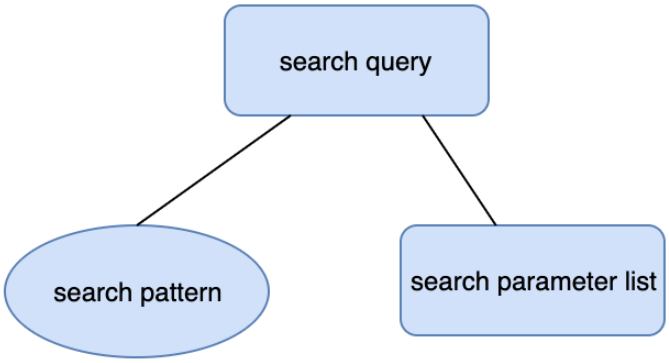

Fig. 2

The diagram uses circular shapes to denote leaf terms (also called atoms), and rectangular shapes to denote non-leaf terms. For simplicity we assume a search parameter list, even though the collection of search parameters is unordered. If we fully expand the terms in the search parameter list, it looks like this:

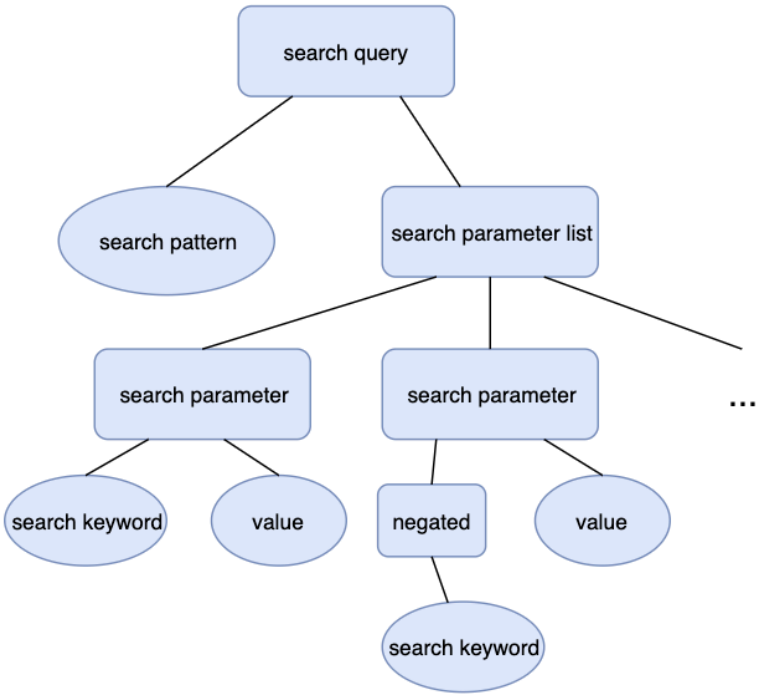

Fig. 3

Note that the above tree (and in general, any abstract syntax tree) does not dictate the data structure implementing it (that is an implementation detail). For example, we currently represent the search parameter list above as a map of **search keyword**s and their **values** in the backend, but this is not important for the discussion at hand.

Search goals

I identify some of our search goals to motivate how they relate to the search concerns and representation above.

1. Sourcegraph aims to lead the space of code search. We aim to accommodate the majority of users ("the 80%"), novice users, power users, and we also aim to innovate new kinds of search and new ways of expressing queries that may affect the majority of users over time. *By definition, we must provide different kinds of search functionality (**search kinds**) to do so.*

2. Different search kinds impose different, and sometimes *incompatible* syntax, design, and search behavior or functionality. *We must have a way to separate the components of search kinds in our query language*. That can happen in a principled way, or an ad-hoc way. We currently do this in an ad-hoc way with harmful effects, and I believe it can be done in a principled way that avoids those effects.

Current challenges by example

These challenges concretely show how the search goals relate to the high level representation.

1. We want to interpret the **search pattern** differently based on what **search kind** is active (e.g., with respect to whitespace, quotation, etc., cf. 13-23 in the current state). What this means practically is that **search patterns** *have their own ways of being parsed and validated.* Conceptually, for each **search kind** we really have the following (cf. **Fig. 2**):

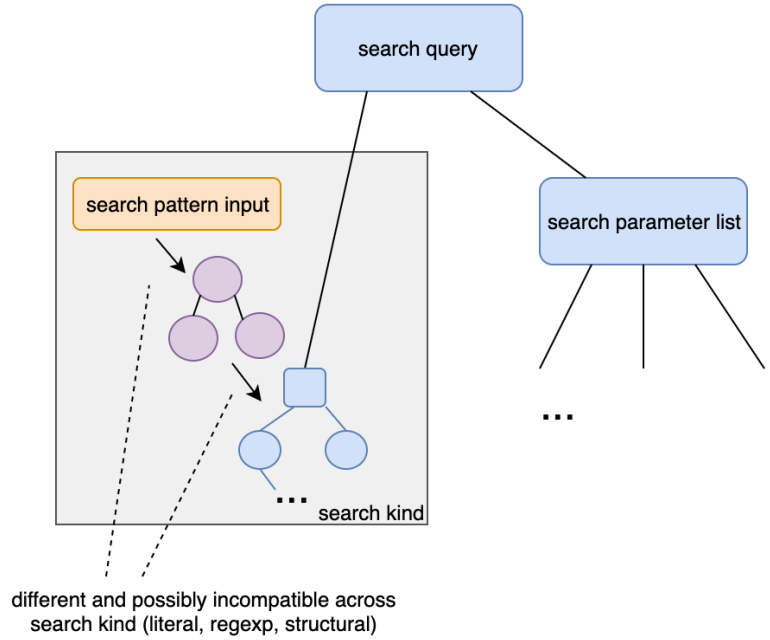

Fig. 4

Challenge 1

The string corresponding to the search pattern input eventually leads to a set of terms that are part of our **search query** for that **search kind**. The implication is that (a) we should not (and indeed, cannot in general) use the same routine to parse and validate search pattern input and (b) when separated, the routines should unambiguously report errors for the active search kind. Not doing the above is one fundamental problem behind [1, 4, 5, 6, 7].

2. We want to parameterize search activities in Sourcegraph with **search parameters**. Our current parameters typically define filters but that does not necessarily have to be the case. We may expand the function and role of these search parameters over time, or create extra rule syntax and semantics that search parameters form a part of. Conceptually, search parameters are really just a low level way of saying that we have our own *Sourcegraph language* for parameterizing search. That means we can define our own language (as in **Fig. 1**). We don't currently define this language formally (or even really informally) outside of the backend implementation, but currently it resembles the "not search pattern input" and associated structure in this diagram:

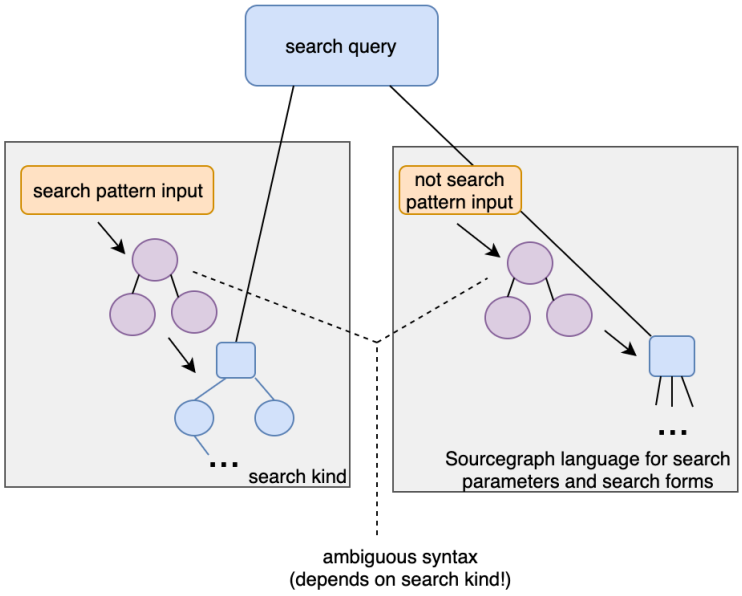

Fig. 5

A search query representation in the abstract syntax tree (i.e., the blue parts forming the final data structure before running search) comprises processed search pattern input (left gray box), and processed input of (what I am referring to for now as) the Sourcegraph language (right gray box). To obtain this final data structure requires parsing the “not search pattern input” string to produce the parse tree. What is considered valid syntax in the parse tree on the right is not the same as that on the left. For example, the parse tree on the right expects search parameters of the form `file:foo` or `file:'foo'`. A string like `file:"` is *invalid syntax* at the parse tree level in the Sourcegraph language.

Challenge 2

A string like `file:"` could be:

a. something that a user wants to search literally OR
b. a genuine typo

As shown in **Fig. 5**, we can’t generally infer the intent because the *syntax is ambiguous*: is this input string part of the search pattern, or a search parameter? We can postulate that one is more likely than the other, and take a default approach, but it is important to realize that this decision has a great influence on expected or surprising behavior--it decides whether to try and group this part of the input string into the (blue) term on the left (and accept the query), or the right (and reject the query). The implication is that *every implicit decision we make to disambiguate these concerns will:*

a. Result in ambiguous user expectations and limit expressibility of queries for different search kinds (a core problem for matching `foo:` and `/foo/` as in [1, 3])
b. Make a tradeoff that loses the ability to express what we want to search for versus precisely validating inputs. I.e., we can ignore the possibility that `file:"` above is a genuine typo and accept it as literal search if parsing fails.
c. Likely entrench us in this decision because changing an implicit behavior later will disrupt user expectations down the line.

Our primary workaround currently tries to disambiguate the above by quoting the search pattern. This mechanism is not sufficient (any longer) because:

(1) In literal search, the act of quoting implies matching literal quotes
(2) The quoting mechanism serves a dual role for regex search where quotes *imply* literal search

Issue (2) leads to the next concern.

3. We want to make it easy for users to disambiguate *search runtime behavior.* A concrete example is point (13) and (18-21) in the current state: For the regex search kind, quoting versus not quoting changes the search runtime behavior. Thus, even for a single **search kind** like regex, we enable users to change these search behaviors through *syntax in the search pattern* (see the different syntaxes we use in 18, 19, and 21 for regex mode)

Challenge 3

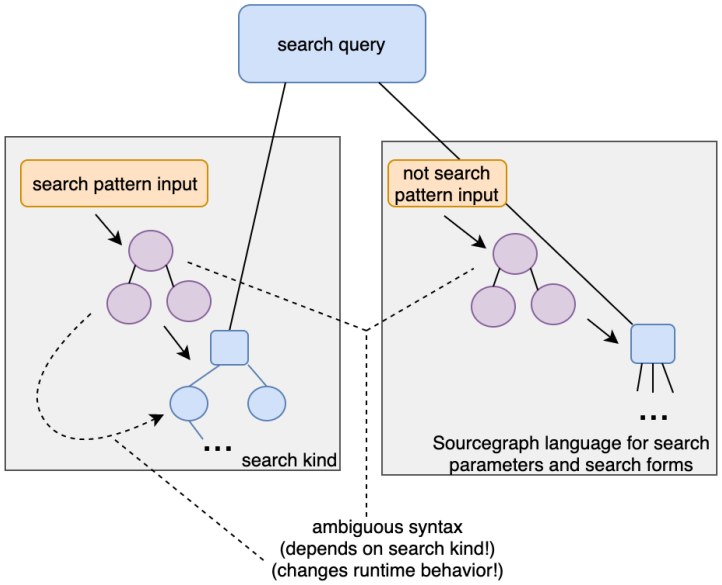

Fig. 6

Using syntax in the search pattern isn't inherently a problem, but combined with Challenge 2 quoting itself is ambiguous because it defines *different search runtime behaviors depending on the search kind* (see **Fig. 6**). We should not use a quoting mechanism to both disambiguate search pattern syntax (Challenge 2) *and* runtime behavior. It compounds the severity and effects of making implicit behavior decisions as above.

4. We want users to define and understand search behavior (i.e., semantics) in a consistent and legal way (i.e., "principle of no surprises"). This concern relates to semantic validation rather than purely parsing the query syntax. One concrete example of this concern is that some of the runtime behavior of search parameters may not be compatible with certain search kinds. For example, `author:foo` has no effect for certain search forms, and semantic validation can check if this is legal or not.

   Another example is that our search behavior and search parameters can (somewhat surprisingly) override existing language specifications. For example, character classes for a regexp pattern that only matches uppercase like `[A-Z]` or `:[[upper]]` will still match case insensitively by default (and can be coerced explicitly by `case:no`). We also say that we support we support RE2 syntax but we apply our own semantics; for example `(?-i)Func` is RE2 syntax for turning on case sensitivity, but we override the regex flag and treat the pattern as case insensitive, unless `case:yes` is specified (i.e., the RE2 syntax here has no effect).

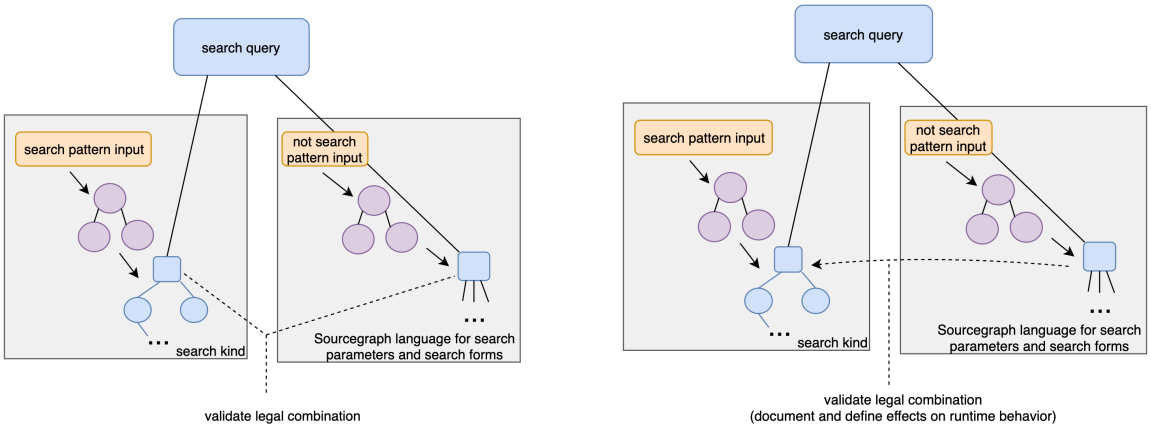

Fig. 7

Challenge 4

As we expand our set of search behaviors we need to implement, validate, and enforce legal combinations of search parameters since these change over time. The implication is that while our Sourcegraph language syntax design can be *language-agnostic* (i.e., we define legal type values for keywords like `case` consistently), its *meaning* can affect the runtime search behavior differently depending on search kinds. For example, structural search is language-aware, unlike `literal` or `regex` search. This means that the `lang:` parameter places an important semantic role for matching parts in e.g., Go versus TypeScript code. As we implement new search (or search and replace!) behavior, supporting the full set of parameters like `file:` or `count:` can take time (e.g., in the case of structural search). The effects and compatibility of these parameters should be easily checkable and gradually expanded on a per-search-kind basis.

Q&A section for search goals and challenges

Q: Aren't the challenges above just a refactoring concern?
A: Unfortunately, no. No amount of refactoring can disambiguate how we currently use syntax to express what and how we want to search (re: parsing search pattern versus parameters across search kinds, quoting, expressing search behavior, and validating it). If some of these problems were alleviated (distinguishing search pattern from the rest of the language), then it is possible to *additionally implement* the parsing and representation across different search kinds. This is not a refactor, but motivates dedicated work.

Q: Why is this worth thinking about or solving now?
A: The complexity of search functionality is increasing, especially with the addition of new search kinds. There are no signs that it will ever decrease (i.e., we are not looking to remove different search kinds). There are various ideas and potential changes for search with respect to UI and user feedback (discussed in the next section). The experience of adding `structural search` as another search kind reveals that it has similar syntactic conflicts as `literal search` with search parameters. `Structural search` will also operate better without quoting or newline escaping. The current state makes it difficult to add or change new search functionality and input processing, and the outlook is that we will run into more of these issues over time.

<If you have questions about this section, consider adding them here.>

Proposal

The following **Fig. 8** is a high level illustration for a **search model** in terms of input representation and language concerns. It informs solutions for the challenges above and future extensions.

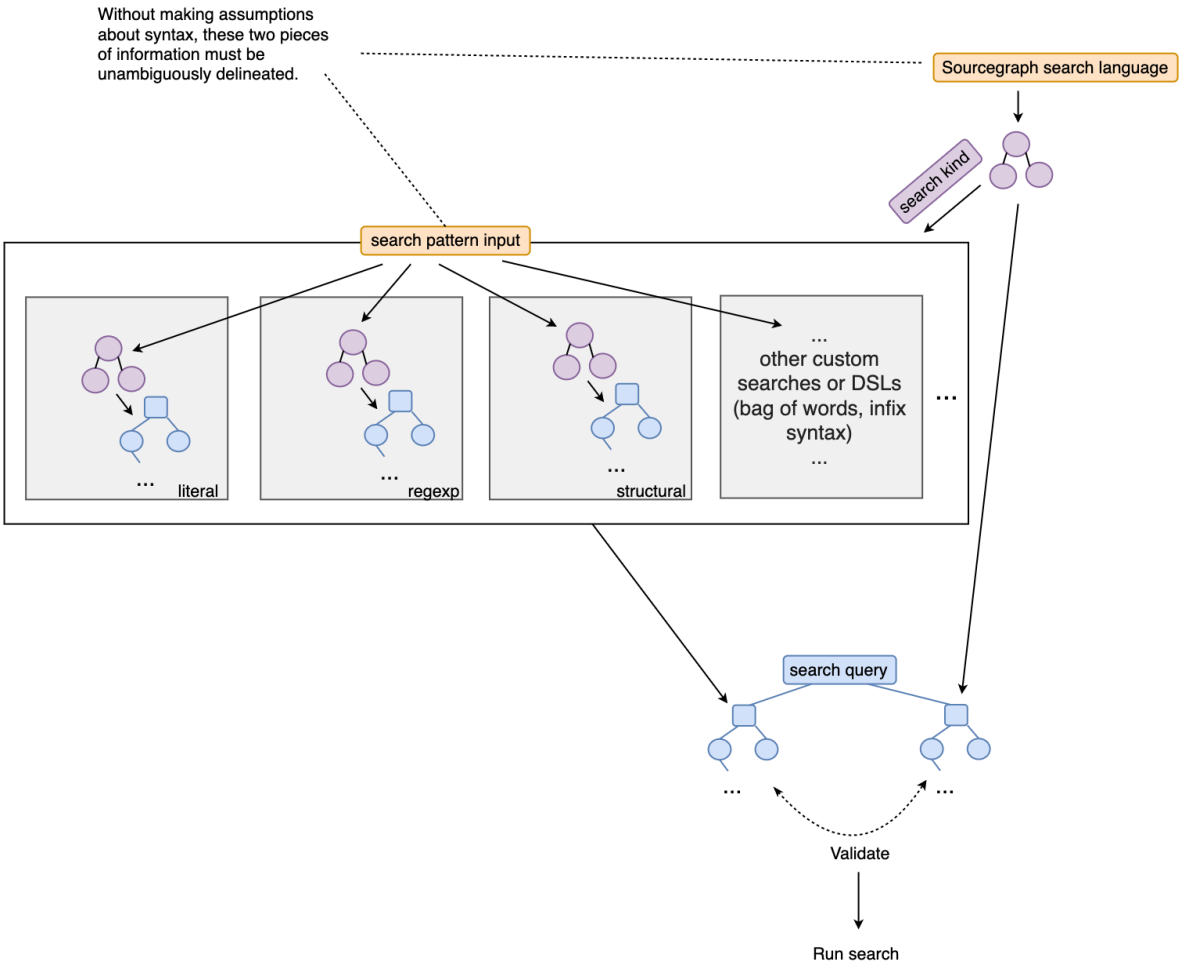

Fig. 8

Introducing a new search kind reveals that the principle complexity we face is that we can no longer easily tell which part of the input is the search pattern, and which part is not (i.e., is part of search parameters and other parts of the Sourcegraph language). Concretely, the introduction of literal search means that

- the previous quoting mechanism for disambiguating these parts is not sufficient (since quotes are significant for literal search)

- the literal search pattern contains syntax that conflicts with parameters (like searching for `foo:`) when not quoted
- The assumptions in the search query scanner/parser no longer hold for literal search (like parsing `` `/.../` `` search patterns for regex).

These are concrete examples for literal search (as in Challenge 2), but bear in mind that these issues are not literal search specific--the same issues would be raised by another hypothetical extension to the search query string, like structural search (and yes, these limitations currently *do* affect the current structural search implementation).

Challenges at the parsing and validation level, and issues that prompt the need to choose implicit ad-hoc tradeoffs (e.g., "what if we just don't surface an error for an invalid-looking field `foo:` since we don't know if the user meant for it to be a search pattern or a filter?") are *all* a trickle down effect of not separating the two orange boxes (search pattern input and the Sourcegraph language) in **Fig. 8**. These two inputs are simply two strings that we need to logically separate before processing.

Advantages of separating search pattern and Sourcegraph language inputs

If we can separate these two pieces of the search query, then:

(a) We can parse the language-agnostic Sourcegraph language irrespective of the **search pattern**. The implementation is forever reusable and extendable without thinking about, e.g., quoting. We can define a grammar for it irrespective of the search pattern input. We can choose any syntax we like without worrying that it will ever conflict with a search pattern. We can independently validate which parts of the Sourcegraph language are compatible with each other (e.g., search parameters), similar to what currently do, while isolating the logic.

(b) We can let (a) decide how to parse the search pattern input string (i.e., select the search kind), *or independently, some other mechanism* (like the UI selector or URL params we use for `patterntype`, etc). The current way we do this using the search input string is a hack and does not have the robustness guarantees that we *can* have.

(c) The search pattern input can be any syntax whatsoever and will not conflict with the Sourcegraph language or other search pattern types, with separate parsers and validations. Search patterns should not and do not need to be quoted to delineate what should be matched.

(d) It becomes trivial to add selectors for other search kinds without worrying about syntax/semantic conflicts, and enforces that we create parsers/checkers for each new search kind without affecting other search kind implementations.

(e) Each search kind can be validated against the Sourcegraph language (i.e., which **search parameters** it supports, etc, and reporting errors) before forming the final search query.
(f) Ambiguous intent and undefined behavior across multiple search kinds for interpolated search patterns in the Sourcegraph language disappear (input like `hey repo:foo search lang:go this` becomes illegal).
(g) We can enable any consumer (webapp, backend) to separate concerns for parsing and validating which inputs they care about, or have the backend be a parser-as-a-service that returns a tree representation of the final search query tree (blue box) in JSON format or similar.
(h) We never have to choose between implicit or ad-hoc tradeoffs, or speculate about their benefit, as far as the search syntax and behavior across search kinds is concerned. We can be precise about what to search and checking the validity of Sourcegraph language terms.

The underlying issues for the long-standing problems and challenges above can all be addressed or enabled by this model.

Long term advantages

There is also a long term view here on our search model going forward. This is important because we have many ideas around modifying search and trying new syntax or languages. Here are just some I am aware of:

- Bag of words search (feedback channel, Beyang)
- Natural language infix operators for search patterns like “foo AND bar” (Christina, if I recall correctly)
- Various text input processing concerns relating to filter (or search parameter) definitions, enumeration, and their value validation (Booleans, Single select, etc.), and API designs in RFC 15. These concerns fit into the language and search model above and plays well with the idea of “interactive mode”
- Interpretation of whitespace in search patterns (e.g., treated insignificantly, as suggested by Dan)
- Rule language to continually refine searches (Quinn, if I recall correctly, and others)
- Structural search with rule DSL (The rule DSL is a separate mini language in Comby)
- Pattern syntax that make use of newlines, and its interpretation (Literal search, Comby)
- Unifying search parameters (or the Sourcegraph language) with search-and-replace/automation. Consider, for example, the use of `scopeQuery` for parameterizing search-and-replace; it shares the concerns above (Rijnard, Automation)
- Search language extensions where rules use and combine with code intelligence (types, j2d) (Rijnard)
- Hierarchical search

- Nested search

Implementing and prototyping the above ideas should be easy and unencumbered as far where they fit into the search input and model (cf. **Fig. 8**), and the proposed model will take us in that direction. The restructuring will also make it possible for clients to pass in the initial set of input strings to the backend and obtain a processed search query, i.e., as an abstract syntax tree that unambiguously expresses search parameters, e.g., in JSON format, so that consumers do not necessarily need to implement any parsing/type checking validation logic.

I propose that we commit to making the implementation changes motivated above, *over time*, that separate the concerns above in the backend (everything below the two orange boxes). This includes defining in writing our language grammar(s), and implementing and restructuring our implementation for search kind parsers/checkers. I'm happy to personally help make that a reality (I'll write the grammars, implement/rewrite the parsers). But to do that effectively: we need to separate the two orange boxes.

Separating the two inputs: the search pattern, and the Sourcegraph language

How can we separate these concerns concretely? That is an open question that this RFC does not propose directly, but here are some considerations. Logically separating the two inputs raises two immediate questions:

1. How and where do we separate the two inputs?
2. How do we support backwards-compatible functionality in the single raw string input?

For **(1):** Defining a data structure for the two string inputs is straightforward, e.g., as a JSON description (this is just an example).

```
{
   searchPattern: string
   sourcegraphParameters: string
}
```

Or even simply

```
{ input: JSON }
```

where JSON has at least two members, one corresponding to the search pattern, and the other the sourcegraph language parameters. The point is that this definition should give flexibility in defining the two objects above, or even objects in the `searchPattern`. One case where it is

useful to have a JSON object in the `searchPattern` is for structural search, which separates the match template and rule (and for search-and-replace, the rewrite template as well).

How this input is constructed is influenced by the user-facing input interface. One possibility is to offer two UI input fields, or to have the UI directly reflect the Sourcegraph language parameters we intend to support (e.g., in interactive mode). I include two relevant comments on having multiple inputs in the UI [Stephen's, Nick's]. As a strawman example, we *could* define a new syntax that delineates the inputs in a single string (like entering braces around the two parts in the search bar `{search pattern} {sourcegraphParameters}`, and defining an ordering), but this is obviously a poor and ugly UX. The natural inclination is that the `searchPattern` should be free of escaping (quotes, or any sort of syntactic delineation). Regardless, the point stands, as discussed at length, quotes are no longer an option.

For **(2)**: To preserve a backwards-compatible representation in a single raw string, I propose we *convert* the current input string to the data structure targeted by (1), once we have settled on one. The *converter function* will use our current backend code mostly as-is and losslessly encode all the brittle, ad-hoc, and implicit treatments we currently perform on the search input string (i.e., the behaviors in 1-21), making them explicit in the data structure in (1). We then *freeze* this converter function and do not add any more logic to it for handling an input search string.

Executing the transition

We restructure the implementation to accept the data structure in (1) at an endpoint as the source of truth for parsing, validating, and running search (this is not a massive change). We mandate that all new search functionality or changes may not go in the converter function. Any *new* search functionality from this point onward must:

(i) ensure that it either targets the data structure (1), *or*

(ii) only operates *on* the data structure in (1), i.e., *after* the converter function may have been applied or

(iii) perform both of (i) or (ii) as required by the new functionality.

We allow clients (e.g., the CLI, webapp, etc.) to directly target the data structure in (1) for search queries, which are passed directly to the backend logic, bypassing the converter function. Any CLI/webapp can use the fallback converter function if it can't or doesn't want to target the data structure in (1). To what extent the GQL interface changes, or whether we allow raw query string to, e.g., have a dual purpose where it can be a JSON value, is open at this stage.

Over time, we restructure, refactor, and implement our search logic to model the components in **Fig. 8**. This can happen at whatever pace we can manage, since during this transition it's always possible to funnel inputs through the converter function to create the data structure in (1). After some time, and with new functionality, the expectation is that user-input is no longer bound for the converter function, but fed directly to the function or endpoint that processes (1).

After some time, we refer to inputs of the converter function as the “legacy search input” representation, and the data structure in (1) becomes the canonical search input representation.

Next steps

Following discussion the above, gathering feedback, and in the event of approval, the next steps are to:

- suggest concrete changes (via follow up RFC) to achieve data structure representation that separates inputs (whether through the UI or GQL interface or some other mechanism). Anyone who feels motivated and knowledgeable about our APIs is welcome to take this up.

- define our Sourcegraph language grammar (in a document) and implement/restructure our search kind components (parsers, checkers) around it. These are concrete work items that require dedicated engineering effort. It does not need to happen all at once or immediately. The intent is that these work items would align with the proposed search model over time. It can be broken up into tasks, and I’m not proposing any engineer allocate their full time to this effort within some timeframe.

Glossary of terms

The glossary defines terms consistently with respect to this document. If you have alternative suggestions for these terms, please leave a comment.

search query - The search input (from a user perspective, the string that goes in the search bar). This document contextually refers to the **search query** both as the raw input string, and its processed form (as a parse tree or abstract syntax tree) for running search in the backend.

search pattern - The expression in the **search query** to match against file contents.

search keyword - The `repo` expression in the string `repo:github.com/sourcegraph`.

search keyword value - The `github.com/sourcegraph` expression in the string `repo:github.com/sourcegraph`. Sometimes abbreviated simply to **value**. Values can have different **types**.

value - see **search keyword value.**

search parameter - A **search keyword** and its associated **value**. A **search query** can contain multiple **search parameter** expressions.

term - an expression or value in the associated language grammar (i.e., the union of terminal and nonterminal grammar constructs).

type - an expression in the **search query** type system associated with a value.

type system - a logical framework of axioms and rules that define legal combinations of **term**s.

search kind - A search procedure for searching file contents, defined by an internally consistent (1) syntax, (2) static semantics, and (3) runtime behavior (i.e., a search kind is disjoint from other search kinds with respect to (1-3), even though they may share traits). We currently support one of `literal`, `regexp`, or `structural` search kinds.

search form - A search procedure for searching contents in software artifacts. Search forms are disjoint, like **search kind**. A **search kind** falls under the **search form** for searching file contents. The `diff` and `commit` searches are alternative **search forms.**

search model - a model depicting dependencies and concerns for search input and processing. I avoid the term “architecture” since an architecture communicates a system that implements (in code) a model, while the converse isn’t accurate.

parse tree - A tree data structure obtained from parsing an input string. All syntax from the input string is preserved.

abstract syntax tree - A validated data structure obtained from an abstract syntax tree. The abstract syntax tree is lossy with respect to syntax (some syntax in the parse tree is throw away).

check - a procedure that validates that a data structure is well-defined with respect to a semantics (e.g., a **type system**).

Notes:

https://github.com/sourcegraph/sourcegraph/issues/6314 affecting
https://github.com/sourcegraph/sourcegraph/pull/7669