RFC 254 IMPLEMENTED: Search query extension: selectors and filter predicates

Editor: rijnard@sourcegraph.com
Status: Implemented

Requested reviewers: Loïc Guychard Stefan Hengl Keegan Carruthers-Smith Thorsten Ball Camden Cheek
Approvals: Thorsten Ball, Keegan Carruthers-Smith, Erik Seliger, Loïc Guychard
Team: Search

Explicit tag for people who might like to selectively read up and provide feedback, not required to do a full review or give approval unless they want to: Juliana Peña Rok Novosel Eric Fritz Stephen Gutekanst Beyang Liu Felix Becker Quinn Keast Olaf Geirsson

Problem

We've seen recurring requests where users want to:

1. select and display only certain kinds of results
2. filter search results by various conditions

While we support various filters and some ways to display results, there are useful functional searches that we don't support directly:[1]

- Find matches for `foo` inside a repository, and only show the matching repositories
- Show file content matches for a pattern inside a repository, but only if that repository contains a particular file (i.e., the file does not have to contain matches, it just needs to exist)

There have been a couple of informal proposals to extend Sourcegraph's query syntax to enrich search capabilities for these kinds of use cases. There are also experimental filters that combine functional search behaviors in ad hoc ways: repoHasFile and repoHasCommitAfter. These generalize poorly, and we can't keep adding ad hoc filters like `repoHasContents`. We need to introduce language constructs to decouple some of these conditional filters and relations.

At the same time, we see requests and opportunities for incorporating richer kinds of (meta)data into code search. For example, "search for `foo` only in comments", "search only in files I own as determined by CODEOWNERS files". The challenge here is to introduce language constructs and syntax in such a way that these capabilities are accessible in the syntax and implementation, if and when we are ready to introduce them.

[1] I say directly because there are indirect ways to achieve some of these. E.g., on the command line by combining src-cli and jq.

Navigating this proposal

There's a lot of discussion around context and rationales for this proposal. If you want to skip some of the exposition, jump to parts of the proposal and you're more interested in:

"What are some of the past proposals/concerns/complaints leading to this proposal?" → Background

"I care about making Sourcegraph search work better with more freeform queries, magical file and symbol completion, and things like ranking. How do those things fit into this proposal?" → Scope

"Which key features are proposed for approval?" → Part 1: Selector syntax and examples.
Part 2: Filter predicate syntax and examples.

"I don't care about background or motivation, show me use cases and syntax" → Look out for a hammer 🔨 in the examples tables of Part 1 and Part 2 above.

"Why this way? What about doing this in another other way?" → Select: motivation
Filter predicate motivation
Rejected alternatives

"Can I propose some alternative considerations and syntax?" → Sure, share your thoughts in Open questions.
For context, keep in mind the points in
Select: notes,
Filter predicate notes,
Rejected alternatives

Background

These are related issues for additional context, roughly in order of relevance. I list them here to convey the scope, complexity, and prior discussions that bear on this proposal. I don't recommend reading through these issues unless you're interested in the history or want to deeply solve some of these issues in a radically different way compared to this proposal.

Group by the repos or files that contain matches: https://github.com/sourcegraph/sourcegraph/issues/12055
Nested searches: https://github.com/sourcegraph/sourcegraph/issues/1005
Incomplete proposal before current and/or operators: https://github.com/sourcegraph/sourcegraph/issues/4774
Search different token types, "in:", "is:" : https://github.com/sourcegraph/sourcegraph/issues/2204

Return only file contents: https://github.com/sourcegraph/sourcegraph/issues/8743
Repo has file with symbol: https://github.com/sourcegraph/sourcegraph/issues/4610
Select only additions of diffs: https://github.com/sourcegraph/sourcegraph/issues/11599
Piping selectors for CSV output: https://github.com/sourcegraph/sourcegraph/issues/1525
Campaigns repositoriesMatchingQuery

Shortlist of relevant language syntax, if you're unfamiliar or otherwise interested in how related domains tackle aspects of querying:

GH search
Bazel and function syntax
CodeQL Predicates.
jq
SQL
MongoDB

Goal

The goal of this RFC is to:

1. Outline the high-level capabilities we want to add and the contexts in which they matter. This is two-part: (1) selecting search results and (2) conditionally refining and filtering search results.
2. Propose and motivate language syntax to express these capabilities.

Guiding principles, please keep these in mind:

A. The additions in this proposal stress **advanced** usage to support capabilities that go beyond what is possible in Sourcegraph today. Allowing more sophisticated searches (increasing expressivity) by necessity introduces more complexity to the query language. This impacts users (learning curve), and ourselves (maintain and implement advanced searches). If this proposal sounds more complex than what you're used to seeing or thinking about when using Sourcegraph, that's expected and part of the nature of the problem :-)

B. The solution outlined here favors a **unified** approach so far possible re: data type definitions, query syntax, expressions, and code maintenance rather than attempting to ad hoc support instances of advanced use cases. You may read some of this proposal and think, "for this particular use case, doing it this new way sounds more involved than it should be, we could get away with just doing X". The problem is that in many of these cases, doing X is an ad hoc idea or shortcut that works only at a shallow level and sacrifices consistent, predictable, and uniform notions. Ad hoc approaches lead to ad hoc comprehensibility for users (difficult to learn or recall syntax), ad hoc documentation (difficult for users to find, difficult for us to maintain), and ad hoc implementation (bugs, difficult to test, difficult to change).

Scope

This proposal focuses on repeated requests for better ways of expressing filter operations on the basic result types that Sourcegraph exposes (repos, files, contents, symbols, commits). Introducing the two mechanisms in this proposal satisfies immediate user desires for more powerful searches. It also lays the groundwork for higher-order search functionality,[2] and more powerful capabilities that stand to benefit campaigns and code insights.

Many folks at Sourcegraph have floated interesting ideas for code search that this proposal does not directly cover. Here are some search use cases that are important, but *not* directly relevant to this proposal:

- Having Sourcegraph interpret more freeform query inputs (e.g., just paste a commit hash and have Sourcegraph try find and show the relevant commit)
- Smarter file and symbol completion
- Ranking search results

Proposal for high-level search mechanisms: selectors and conditional filtering

Ignoring syntax for now, there are two high-level capabilities that we should aim to support generally. They are:

Part 1: A select filter selects a kind of result. I.e., we seek a mechanism to select subsets of data on a result set, where a result set is a subset of the domain of our inputs.

Part 2: Conditional filtering is the notion that evaluating a **functional predicate**[3] returns a result set on some condition, where the condition is a general notion. The term functional predicate is perhaps overly general and mathematical, and detached from Sourcegraph as a search tool. So in the rest of this RFC I'll synonymously refer to functional predicates as **filter predicates**.

> Here's a concrete example to demonstrate how filter predicates fit into more sophisticated search queries that user's have requested in the past:
>
> > Show file content matches for a pattern inside a repository, but only if that repository contains a particular file (i.e., the file does not have to contain matches, it just needs to exist).

[2] More context on what I mean by "higher-order search functionality" to come in a follow up RFC, stay tuned.
[3] For practical purposes, think of a functional predicate as an ordinary function in your favorite language that takes some inputs and produces an output. In the context of this RFC, a functional predicate is different from "just a function" in that it is defined over certain inputs that are particular to Sourcegraph, and the domain of code search and code search use cases.

Ignoring syntax, we could define a filter predicate **contains** that takes as input a set of repositories and a file pattern, and returns the set of repositories if a “contains” relation holds, i.e., that the repository contains a file satisfying the file pattern:

```
contains(set_of_repositories, file_pattern) → set_of_repositories
```

A filter predicate is thus an umbrella term for describing a mapping function that evaluates a boolean-valued expression on the domain of our inputs to yield a result. The domain of our inputs are defined by the various data that Sourcegraph stores (repositories, file paths, file contents, commits, symbols) and domain-specific data types that these functions accept (search pattern, etc.).

Both **select** and **conditional filtering** capabilities are ultimately operations on the result set of a search. At the implementation level, we can delegate these operations to a so-called **Rule Engine** that does processing on the results of a search that you can run on Sourcegraph today. I.e., these new capabilities are an optional processing step based on a syntax extension of today’s query language. There are no proposed changes that break any part of Sourcegraph’s current query syntax.

Result set definition

It’s useful to define a concrete data type that selectors and conditional filtering operate on. A result set of these operations is defined by at least the following data types, which correspond roughly to the result types in our GraphQL schema:

```
repository
|
|- file
|   |
|   |- content
|   `- symbol
|
`- commit
    |
    |- message
    |- diff-content
    |- author
    |- date-time
    …
```

In practice there are additional data types to think about, and a more complete picture of what we might like to expose for filtering and selection, is e.g.,:

```
repository
|- name
|
|- file
|  |
|  |- name
|  |- content
|  |  |
|  |  |- line
|  |  |- comment
|  |  |- string-literal
|  |  |- code
|  |  ...,
|  |     |
|  |     `- range
|  |        |
|  |        |- lsif-hover
|  |        |- lsif-declaration
|  |        |- lsif-references
|  |        ...
|  |
|  |- language
|  `- symbol
|     |
|     |- variable
|     |- const
|     |- package
|     ...
|
|- commit
|  |
|  |- message
|  |- diff-content
|  |  |
|  |  |- comment
|  |  |- quoted-string
|  |  |- code
|  |  ...,
|  |     |
|  |     |- added
|  |     |- deleted
|  |     `...
|  |
|  |- author
|  |- date-time
|  ...
|
|- issue
|  |
|  |- comment
|  ...
```

```
|
...
```

The point is that there is a decomposition of data that we can filter and select today, and ones that we should explicitly think about for future extension (e.g., scoped naming for symbol tokens, LSIF data associated with ranges of file contents, comments related to issue trackers, etc). This proposal aims to reference a consistent definition for supporting new additions for filtering/selection over data.

Part 1: Select

Behavior

For any result set defined in terms of the data types above, a select operation extracts those data members (including useful properties of parent definitions, i.e., content matches return file path names and repo names). This solves many user requests along the lines of “give me only the repos I’m interested in if it matches this query”, see #12055. Selecting multiple disjoint data types merges into the same result set (e.g., where `commit` splits to separate branches for commit `message` versus commit `diff-content`.).

Selector syntax and examples

The proposal is to add a `select:` parameter that accepts values in the result set definition. I think the importance and flexibility of a select operation at the toplevel of a query, and as a new parameter, is well-justified. Here are examples that we solve for user requests, and interesting ways to combine and use `select:` for future extensions. None of these behaviors are currently explicitly expressible.

The **Query** 🔨 column shows queries that will be imminently expressible and easily implemented. I.e., “we can add this very soon and with little risk, I’d like approval for this”.

The **Forward-compatible queries** 💡 show aspirational ways to easily extend deeper ways to use the syntax. I.e., “We can add this soon too, but gradually depending on the result kind (e.g., symbol, versus diffs, versus…). These are subject to more discussion, priority, and effort and not explicitly approved in this proposal.

Query 🔨	**Description**
`repo:foo file:bar baz select:repo`	Adding `select:repo` simply converts the results of a traditional query to repository name results. See #12055 for requests. This is also a user-facing syntax and solution for what campaigns functionally does with the `repositoriesMatchingQuery` field in campaigns spec.
`repo:foo file:bar baz select:file`	Select only file results for files whose name matches bar in

	`repo:foo` and containing pattern baz
`(repo:foo or repo:bar) file:Dockerfile select:repo`	Select works over result sets of expressions (merges results of the same kind)
`repo:foo file:bar baz` `(select:repo or select:file or select:content)`	Select multiple disjoint result kinds using an or-expression. In fact, the default Sourcegraph search selects multiple result kinds corresponding to an implicit expression `(select:repo or select:file or select:content)`. Select gives explicit control.
`file:Dockerfile select:repo`	Equivalent to the query `repohasfile:Dockerfile`. Note, though, that select is not powerful enough to match the expressivity of `repoHasFile` for queries that also contain search patterns for file content. More on that in this interaction in the **Conditional Filtering** section. This use case solves some of the examples in #1005 and subsumes some of the need for `repoHasFile`.
`type:commit after:"last thursday" error select:repo`	Match `error` in a commit message after a specified time. Removes the need for `repoHasCommitAfter`. And is equivalent to current `repoHasCommitAfter` usage. By functionally equivalent I mean that the query gives the same expected result if we don't think about how the result is computed. To compute this efficiently we would likely optimize the query when we see, e.g., `select:repo`.

`type:commit author:rob select:repo`	A query to handle the request “Users should be able to quickly return a list of repositories they have committed to. Preferably by date descending.” #13236. We are currently limited by commit/diff searches in computing this result in the backend. This is just demonstrating the query that would give the desired result.
Forward-compatible queries 💡	**Description**
`type:symbol Search (select:symbol.class or select:symbol.function)`	Specify only result kinds in the result definition. This would achieve related functional behavior discussed in #2204.
`type:symbol Search select:class,function`	Alias of the above, but more terse. Comma-separated values for select can be defined to mean “`or`”, and the result names can be inferred from or qualified by `type:`.

Notes for select parameter

- **An alternative syntax we can consider to `select:` is `return:`.** I think `select:` is clearer because the operation is data-driven, and on a result set, rather than a “function”-like operation with control flow properties, where something like “return” is used. Select is also a familiar term in data-driven queries popularized by SQL and so on.

- **The expressivity that `select:` adds makes `type:` redundant in many (but not all) queries.** For example, `type:symbol` is potentially redundant in the last two examples in the table. Deprecating or rethinking `type:` is something we can defer for now. `type:` can continue to exist and operate the way it currently does, and is compatible with `select:`.

Motivation

Every search query today performs an implicit select operation, but we’ll enable more use cases in allowing queries to select more than one implicit result type. Implicit select examples:

Query	Description
`repo:foo`	Returns only results for (i.e., “selects”) repos, and not files or commits, since files or commits don’t

	intuitively make sense to return.
`repo:foo file:bar baz`	Selects matches for `baz` in files in repos, or files containing `baz` in the file name, or repos containing `baz` in the repo path.
`type:repo foo`	Alias for `repo:foo` in current behavior
`type:symbol foo file:bar`	Perform a symbol search for symbols matching foo in file names containing bar and return symbol results.

Observe that the `type:` parameter currently influences the selected (or returned) result type. E.g., `type:symbol` is the only way to get symbol results back in Sourcegraph. But `type:symbol` is dually used to tell Sourcegraph to run a search for symbols. So, the `type:` parameter currently defines both the INPUT and OUTPUT kinds of a search. It acts like a function signature that implies a default result kind to select in the OUTPUT, but doesn't let a user change this OUTPUT kind.

Sometimes the default OUTPUT kind is sensible, but the current state is limiting (e.g., what if we wanted to select only file paths containing a symbol). Our result tabs also suffer from this issue, and adding `type:` may change the meaning of a search query rather than just filter the result kind. If we have a way for queries to express searching for *contents* inside repositories, or *files* inside repositories, and then just have a way to filter by the kind of result they want (file, repo, …) we end up with a more expressive API, and also make it possible to build better UI filtering components at the API level than doing result filtering in the client.

Part 2: Conditional filtering with filter predicates

Behavior

A filter predicate returns a result set on some general condition that evaluates to true or false. Filter predicates enable sophisticated conditional filtering that test certain data relations in a result set, and can then produce a result set where those tests (relations) are satisfied. Here are examples of desired behaviors:

- Search a repository across all files for the pattern P but only if the repository contains at least one match for the pattern Q in any file.[4]

- Show file content matches for a pattern inside a repository, but only if that repository contains a particular file (i.e., the file does not have to contain matches, it just needs to exist).[5]

These behaviors rely on a general relation where a repo *contains* some data, i.e., "a repo contains file contents matching the regex pattern P". We can define a "contains" relation formally in terms of a subset relation, but in this RFC we'll just appeal to the intuitive definition. Filter predicates allow a straightforward way for users to express these high-level queries. Of course, we'll define and implement the code for checking that "X contains Y" for valid inputs of X and Y.

[4] This is a recent request from a user.
[5] Based off of examples in #1005.

Filter predicate syntax and examples

The proposed syntax of a filter predicate is similar to a call in many languages:

name(*n*, *m*, ...)

A filter predicate has a name and arity that accepts some parameters *n*, *m*, …. Predicates take any string as arguments and are interpreted by the underlying implementation. Standard quoting and escaping apply for the reserved syntax: parentheses and commas.

A filter predicate's *name* is registered in the query language (so that we can detect whether such a predicate is supported), and it is registered to an existing filter (e.g., registered to the filter `repo:`). Filter predicate syntax may only follow filters to which that predicate has been registered, e.g., you will only see filter predicates syntax after a filter, as in:

`repo:`*name*(n, m, …)

The **Query** 🔨 column below shows queries that will be imminently expressible and easily implemented. I.e., "we can add this very soon and with little risk, I'd like approval for this".

The **Forward-compatible Queries** 💡 show aspirational ways to easily extend deeper ways to use the syntax. I.e., "We can add this soon too, but gradually depending on the result kind (e.g., symbol, versus diffs, versus…). These are subject to more discussion, priority, and effort and not explicitly approved in this proposal.

The **contains** filter predicate enables use cases where users want to first filter repositories or files by some search query, and then search in the result of that operation.

Query 🔨	**Description**

`repo:contains(file:foo) bar`
`repo:contains.file(foo)`
	Show matches for `bar` in repositories that contain the files matching `foo`. `foo` can be any valid `file:` value, like a regular expression.

`repo:contains(content:foo) bar`	Show matches for `bar` in repositories that contain some file contents matching `foo`. `foo` can be a valid `content:` value. We decide on a default pattern type e.g., a regular expression or literal by default. Additional predicate filter params may

	override the default.
`repo:.*sourcegraph.* repo:contains(content:foo) bar`	As above, but restrict the repository scope to repositories that contain the word `sourcegraph.` I.e., predicates are compatible with existing filters that take plain values.
`repo:contains(content:foo) or repo:contains(content:bar) baz`	Search for `baz` in repos containing either foo or bar. I.e., filter predicates work across operators.

`repo:contains(file:foo) and repo:contains(file:bar)`	Finds repositories that contain both the file `foo` and the file `bar`. Note this useful query is *not* expressible in Sourcegraph today. A similar looking query we can currently write is:
	
	`repo:.* (file:foo and file:bar)`
	
	which is equivalent to:
	
	`repo:.* file:foo file:bar`
	
	The behavior of the above query, for whatever historic reason, only finds files in any repository where a single file *name* contains both `foo` and `bar`. I.e., the and-operation intersects on substring containment and not whether repo's contents contain both a file `foo` and file `bar`.

`repo:sourcegraph repo:contains(file:\.py) file:Dockerfile pip`	Equivalent to our documented `repoHasFile` usage when the query includes a pattern `pip` to match. Example query. The `contains` filter obviates the need for `repoHasFile`, and can also be used with `content:`.
Forward-compatible queries 💡	

`repo:contains(file:README content:foo) file:bar baz`	Searches for `baz` in files matching `bar` in repos that contain a file matching `README` where those matching README files have matches for `foo`. The `contains` argument here resembles a search query, but it need not (predicate arguments can take a string shape of whatever form). This syntax is to demonstrate how we could allow a use case of "run a traditional search query but only on repositories where a specific file X has matches for Y" by allowing the `contains` predicate to accept expressions itself, including `and` or `or` operators. This use case is considered an immediate piece of follow-up work for `contains`, but one we can punt on for now.
`repo:contains(symbol:bar)`	Implements a `contains` filter predicate that relates repos to symbols. Filter repos containing a symbol `bar`
`repo:contains(symbol.function:bar)`	Implements `contains` to accept more granular symbol kinds (function, comment, etc.).
`file:contains(lsif.hover:SearchResultResolver)`	Filter files with LSIF hover information match the string `SearchResultResolver`. Can express, e.g., a cross-repository search on the "Graph" of code by hooking into LSIF semantic data and not just text based.

Notes for **contains** predicate

1. **An alternative name for `contains` is `has`.** Although `has` is shorter (nice), I feel it's less precise.

2. **In general, forward-compatible queries can implement `contains` on any result type in the result set definition**, and expressions on these. The `contains` predicate is thus defined on file and content for repo, and on content for file. We can expand it for commit, etc.

3. **Alternative syntax considered and rejected**. See later discussion of alternatives that considers filter predicates more generally.

4. **Relation to piping.** Piping data to a subsequent command is a familiar mechanism in Bash scripting, and built into tools like jq. Over time, I've realized that the majority of use cases and requests we've talked about are a subset of a general pipe operation, and that adding a general pipe operation adds more constraints and complexity than we need to enable these use cases. I'm going to elaborate on just some of these points, it's not exhaustive. On constraints: a pipe operation is *ordered*. This is a semantic behavior irrespective of syntax, so I'll only explain why I think ordered expressions in our queries are best avoided until we identify more compelling reasons. Ordering imposes that users sequence commands in a valid way. Our query language currently is completely unordered, which I've come to appreciate for its simplicity (just add `foo:bar` when you want to modify the query). Requiring users to consider ordering means that we have to communicate which expressions are valid in these ordered operations (does it make sense to pipe X to Y?) and introduce a very particular syntax for this behavior. On complexity: unlike Bash scripting (resp. jq), Sourcegraph doesn't implement a routine to evaluate a pipeline of operations on, essentially freeform text (resp. extremely simple JSON spec). It would be a significantly complex undertaking to build such a pipeline into our backend, and ensure that some piping expression is valid. Piping expressions are less constrained from a behavioral perspective than atomic filters like `repo:foo`, making it more difficult to implement efficiently. Conversely, filter predicates gives the user a declarative interface where they don't have to explicitly think about ordering, and we can implement the backend in a way that optimizes constrained operations rather than trying to efficiently evaluate a generic pipeline of commands.

5. **Relation to and-operators.** An and-operation semantics evaluates the intersection of two results sets (e.g., if I have a set of files A that match X and a set of files B that match Y and I want the set of files that contain X and Y, then those files are in the intersection of sets A and B). You can think of an and-operation as an *unordered pipe* operation. That's because taking the intersection of two result sets is taking the subset of a given result set. Piping also reduces a set to a subset, but the expression is an explicitly ordered operation. Conversely evaluating the result of an and-operations are not explicitly ordered. For example, taking our previous set A with matches of X, we can find the intersection of A and B as a subset of files in A that match Y. Or, equivalently, we can start with the previous set B with matches of Y, and find the intersection of A and B as a subset of files in B that match X. This unordered property makes and-operators convenient from a user perspective (the user doesn't care if we first find files with X or Y, so they can order it either way) and from an implementation perspective (we choose how to evaluate finding matches and can optimize on the properties of our data). In this context, the filter predicate `contains` is a subset operation based on some relation (taking a subset is analogous piping a result through a filter). Because this subset operation produces a result set, the result set can be further evaluated in the context of other expressions (e.g., an and-operation). This combination of "unordered piping" using our existing and-operations, along with subset operations based on relations using `contains` compose to give expressive power that is equivalent to evaluating pipe operations without (for now) introducing the semantics of arbitrary piping. For example `repo:contains(A) and repo:contains(B)` is like a pipe operation that checks that a repo satisfies two `contains` relations

but without requiring an ordering (like checking A first and then piping the result to checking contains B next, or vice versa).

Additional filter predicates

We have an immediate demand for implementing the `contains` predicate. The next couple of examples are compelling use cases that we do not need to immediately implement, but open up a lot of possibilities and motivate the predicate syntax. These are all **forward-compatible queries** 💡

The **commit** filter predicate filters repositories or files to search based on some commit property. Note that this is unlike a *commit search*. This predicate enables searching the *repository or file contents* based on whether that content is part of a commit.

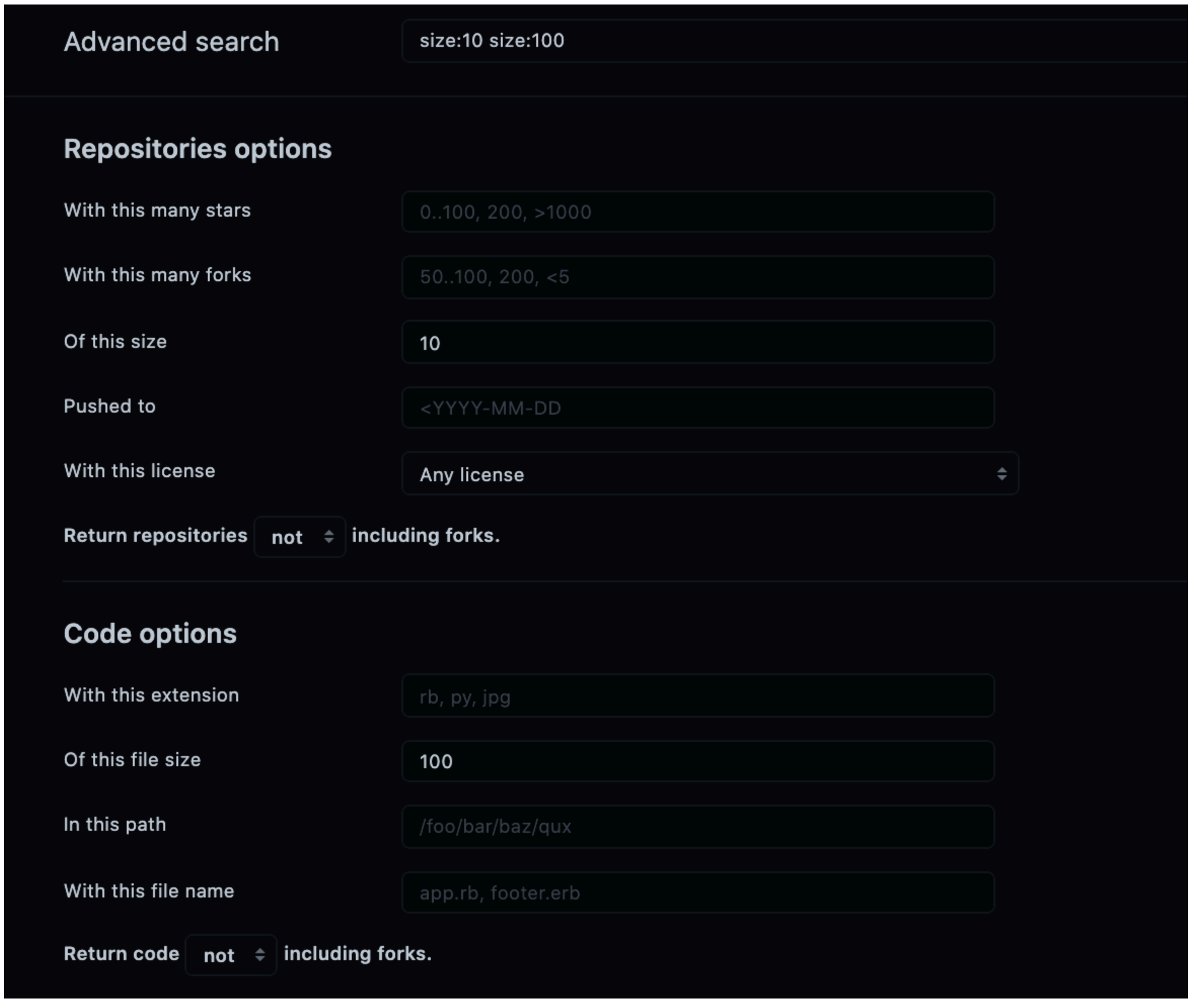
`repo:commit(after:"last thursday") error`	Searches for `error` in repositories (not commit data) where the repository has had a commit after `last thursday`. A useful construct for bisecting code changes based on some pattern.
`file:commit(after:"last thursday") error`	As above, but filtered to files satisfying the commit time.

The `commit` predicate can be extended to accept other parameters for commit data in the result set definition, similar to commit searches (e.g., author, message, …).

The **size** filter predicate filters data by size (for supported types in the result set definition).

`repo:sourcegraph file:size(>=1MB)`	Show large files (>=1MB) in repos matching sourcegraph.
`repo:sourcegraph repo:size(<10MB) error`	Search for `error` in repos that are smaller than 10MB in repos matching repositories.
`repo:sourcegraph file:size(<1MB) error`	Search for `error` in files that are smaller than 1MB in repos matching sourcegraph.

Here it's important to point one reason why filtering is better expressed as filter predicates than introducing a new toplevel filter like `size:...`. Properties like size are shared across different (i.e, disjoint) types of data, like repositories and files, for instance. Notice how the GH query builder will add `size:10` and `size:100` to the same search query for different types (repositories versus file sizes), and this is problematic and ambiguous:



Advanced search: size:10 size:100

Repositories options

With this many stars	0..100, 200, >1000
With this many forks	50..100, 200, <5
Of this size	10
Pushed to	<YYYY-MM-DD
With this license	Any license

Return repositories [not] including forks.

Code options

With this extension	rb, py, jpg
Of this file size	100
In this path	/foo/bar/baz/qux
With this file name	app.rb, footer.erb

Return code [not] including forks.

There is more than one way to resolve this ambiguity for our use case. For example, grouping together toplevel filters would allow us to infer, in some cases, whether a toplevel `size:` refers to file sizes or repo sizes:

`repo:sourcegraph (file:.* size:<1MB)`	**Straw Man example.** Equivalent to proposed syntax: `repo:sourcegraph file:size(<1MB)`

Here we would need to write some query processing code that infers `size:` refers to `file:`. From a syntactic perspective, the `file:.*` is awkward (but necessary) if a user wants the filter to apply to all files. This is the same reason that I'm inclined to think that if Sourcegraph had implemented a size predicate similar to other ad-hoc parameters, it wouldn't be a filter like `size:`, but would rather have taken the form `repoHasSize:`, which is awkward.

A more problematic case is that this hypothetical inference routine will not work on the following query, using the same syntax, but with the intent of also putting a constraint on repo size:

`repo:sourcegraph size:>10MB (file:.* size:<1MB)`	**Straw man example.** Impossible to interpret correctly, but could be taken to *intend* the unambiguous and correct meaning of the proposed syntax: `repo:sourcegraph repo:size(>10MB) file:size(<1MB)`

The above straw man query does not work, because search subexpressions imply scope and this is a desirable property. For example:

```
repo:sourcegraph (file:foo or file:bar)
```

Implies that we only search for files `foo` or `bar` in repositories matching sourcegraph. If `size:` is introduced as a toplevel filter, it could imply that it scopes subqueries. Does `size:>10MB` then override the subexpression `size:<1MB` for files above? Or does the subexpression override the toplevel filter? Answering these questions introduces new and wonderfully complex semantic behaviors that will make life very unpleasant for users and our engineers. The intent of the converse query is immediately clear with filter predicates (and far easier to implement the query processing):

```
repo:sourcegraph repo:size(>10MB) file:size(<1MB)
```

Besides clarity, we remove the need for excessive parentheses and awkward additions like `file:.*`. The only arguable downside compared to the straw man query is that `repo:` is repeated for the filter predicate.

The **time** filter predicate filters results based on temporal data. For example, we've had code insights request for "search repository contents on a specific date" #10820. There are a couple of ways to name or parameterize this predicate, so these are rough examples. Like size properties, temporal properties also hold over different kinds of data. So, whatever naming or parameters we give time filter predicates, it is similarly defined for different types of data (e.g., repo, file, etc).

`repo:sourcegraph repo:on(2021-01-01) error`	Search for `error` in repository contents on a certain date.
`repo:sourcegraph file:created(>2021-01-01) error`	Search for files created after some date.

GH search exposes various filters for temporal data, like `author-date` that I think are aptly decoupled with filter predicates like `author:date(<2021-01-01)`

The **stars** filter predicate demonstrates how predicates contextually enable filtering depending on codehost/metadata without polluting our toplevel filters.

`repo:stars(>100) error`	Our backend implements a routine to filter repositories by, say, GitHub stars, if we store that metadata. This predicate is only valid in certain contexts, e.g., for OSS/cloud repositories on GitHub where we collect the metadata.

It's useful to think about diversifying our search capabilities to metadata like GitHub stars, but in a more sustainable way than introducing toplevel filters like `star:` which do not matter in, say, customer environments.

The **group** filter predicate demonstrates how predicates can help cut down on increasing toplevel filters.

`repo:group(CNCF) error`	Equivalent to `repogroup:CNCF` but without introducing the need for a toplevel `repogroup:` filter. On its own, it can help with some query processing and establish `repo:` as the definitive filter for repositories. But `group` is better motivated by the next example in conjunction with `file:`.
`file:group(CHANGELOG) regression`	Search all files associated with changelog entries across all repositories by using `group` on the `file:` filter. The `CHANGELOG` group is of course defined by some pattern, similar to what we do for `repogroup:`. For example, changelog files matching `/.*changelog.*/i`. With `group` as a predicate though, there is no need to consider introducing something like `filegroup:`. Of course, the group of files can be specific to a team, some specific functionality, and so on.

These examples are just to give a flavor of possibilities for initial filter predicates. They are "primitive" in the sense that the data they operate on has straightforward relations (`contains`, `size`, `commit` data associated with other data). The last example using `group` gets closer to the idea that predicates open up higher-order possibilities, and if that interests you, stay tuned for an upcoming RFC on *Code Search Built-ins.*

Motivation

The need for more sophisticated filtering is clear from #1005 through to recent requests for search. Campaigns also stand to benefit: [6] Campaigns already stand to benefit heavily from the efficiency of using search to power changesets. Users may find they can express finding and filtering repos and files directly using predicates, without (or with less frequent) script-based piping or filtering.

[6] This modulo the work to return "exhaustive" results. The point is that once search is an appropriate choice for exhaustive result fetching (e.g., fetching repos or files matching a criterion), search queries can directly substitute for scripting.

Filter predicates aims to be the vehicle to deliver specialized search behavior and are proposed with these desirable properties in mind:

Users do not need to understand or learn an overly general or heavyweight language or syntax. Here it's useful to compare functionality we aim to provide to a query engine like `jq`. While `jq` is extremely powerful,[7] it is also complex (it understands data structures in JSON, i.e., objects and arrays, and a user would need to be familiar with the data definitions and JSON structure) [1, 2]. As much as I like the power of something like `jq`, I've come to learn that Sourcegraph usage generally boils down to simple cases like find me X (string, symbol, …) in Y (file, repo, commit). User requests for behaviors that build on this with conditions that resemble "piping" can be made more accessible and declarative by filter predicates without, e.g., an explicit pipe operator (see concrete examples (4) and (5) in Notes for the `contains` predicate). The user requests and use cases we've seen and discussed does not (yet) motivate excessively powerful or complex operations on the data Sourcegraph exposes. While filter predicates are not the end-all-be-all, I can say with confidence that filter predicates fill a gap where strictly more sophisticated behaviors, aligned with user desires, become possible in a straightforward and unobtrusive way. More complex operations can be considered as they come to light.

Small and extensible footprint on the current query language. Filter predicates consist of a name and arity, and are valid for one or more existing filters (like `repo:` or `file:`). By registering a filter predicate name and some simple validation, it's easy to express and retrieve user inputs and implement a backend routine to compute some custom behavior. It gives us a unified way to implement query autocompletion, highlighting, and hints for more freeform inputs. For example, an N-arity predicate can take N inputs, while our current filters like `repo:` accept only one input (the pattern).

Compatible with expressions in the current query language. Once a single filter predicate exists, it's generally ensured to "just work" when combined with and/or operators. Meaning: the syntax and behavior is valid and expressible, and we can tell users "yes that's possible with Sourcegraph". Of course, I'm not saying that the performance of these queries is automatically great, backend implementations will have to consider query optimizations for predicates that are expensive to compute across expressions. See the size predicate straw man examples to understand how predicates avoid interacting poorly with search expressions.

A mechanism to implement tailored and experimental search experiences in specific contexts (Cloud, different code hosts) and separating it from basic search functionality that works across all Sourcegraph instances (see stars predicate).

A declarative framing for search behavior. Linguistically, predicate names anchor behavioral descriptions of use cases: "I want to find a repository that *contains* X" follows from `repo:contains(X)`.

Ease of use through query intelligence. With the help of highlighting, autocompletion, hovers, and autofix suggestions in queries, I think using filter predicates will feel "natural" despite it being a more "advanced" construct.

[7] I like jq.

Constraint

Users cannot define their own predicates. One benefit of filter predicates is that we can define how to run the computation in the backend, and optimize for performance. We should listen to user requests and try to support use cases. The only way to enable user-customizable predicates is with a dedicated framework that constrains the data types and computation that a user can work with, and is outside the scope of this proposal.

Syntax alternatives considered and rejected

At least these syntax alternatives were considered and rejected.

Query	**Idea and reason for rejection**
`repo:sourcegraph contains:X`	**Idea:** Toplevel predicates for testing relations like `contains:` and `size:`. **Rejection:** Does not play well with subexpressions. It would also cause excessive need for parentheses to group a toplevel filter like `contains:` with the associated filter like `repo:`. May need to introduce further toplevel filters like `contains-file:` or `contains-content:` which is only incrementally better than `repoHasFile`
`repo:foo where repo contains file:bar`	**Idea:** introduce a keyword `where` followed by a filter predicate expression. Readable and separates data from filter operation. **Rejection:** Awkward syntax for common search use case if there is no `repo:` filter. I.e., we want to implicitly search over all repositories. If we do not specify `repo:` our search query is an awkward dangling `where repo contains file:foo`. In this case, allowing to omit the `where` complicates query processing. Alternatively, imposing an explicit `repo:.*` before `where repo contains file:foo` feels silly. In general complicates query processing when introducing new terms like `contains`
`repo:foo where repo:contains(file:foo)`	**Idea:** Slight variation to above that cuts down on query processing complexity for `where` clauses **Rejection:** Still awkward for searching over all repos in `repo:.*` case. `where` keyword feels redundant.
`repo:(file:foo select:repo)`	**Idea:** A minimal subquery-like syntax to express the subset result set to search over, in part using `select:`. E.g., this example would mean searching

	over repositories where the query `file:foo` returns a non-empty result. **Rejection:** Unclear semantics, does not immediately convey that this query expresses, e.g., "contains" behavior. Gives a potentially false impression that we can efficiently evaluate subquery searches.

As far as thinking about conflict with existing syntax, there's no real issue here. Predicate syntax will be validated by name, so collisions with literal meanings should be rare. When there is a conflict, we support quoting as an escape for literal meanings across all filters.

Open questions

Share your questions in this section if the proposal doesn't answer them. After thinking a lot (a lot) about this problem and distilling a solution and syntax that satisfies our use cases, I personally do not have many open questions around these choices. Here are a couple of things I'm still thinking about, so feel free to share your thoughts on it here:

- Should predicate syntax be prefixed by a character? For example:

```
repo:+contains(...)
repo:#contains(...)
repo:\contains(...)
repo:%contains(...)
repo:!contains(...)
repo:?contains(...)
repo:@contains(...)
```

The main benefit is from a query intelligence perspective: We can trigger better autocomplete and validation than if a prefix is absent. For example, if a prefix is absent, we may need to think a little about distinguishing predicates from repository paths in suggestions.

- Keegan - Can we sketch out an intermediate representation/query plan for this new syntax? IE for a lot of these queries they can be efficiently executed by the respective backend (eg zoekt could do the filtering for most of the examples). By nailing down the representation for machines we also have nice unambiguous ways to explain what a query is doing. General feedback: This looks good to me. Initially I found the filter predicate syntax confusing and thought we could get by using (select:) on subexpressions. However, your proposed features like size/etc are really cool and make me like it a lot :)

Implementation notes

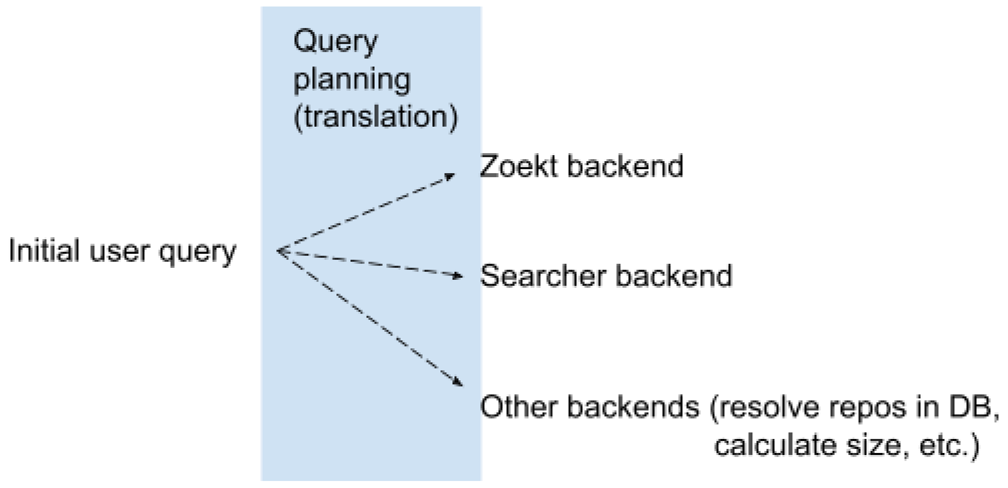
Backend Implementation

Backend implementation relies on additional processing for `select:` and `contains`. Select is rather straightforward to implement, since we can directly filter result types once a search completes. Implementing `contains` means adding general functionality for validating predicates, defining their inputs, and implementing the behavior. Implementing the behavior of `contains` functionally boils down to the same way we currently implement things like `repoHasFile,` but where we now have a better framework for thinking about predicates to anchor the code structure.

Query evaluation, optimization, and translation.

Once a query is in the backend, we decide how to evaluate (compute) a result, and can take liberties with optimizing those computations. For example, the `contains` predicate can be implemented in part (or in whole, I think) natively in Zoekt, as a Zoekt query, when a repository is indexed. When it is unindexed, we might evaluate `contains` in a different, albeit slower way (say, by using unindexed search). This notion is general and not specific to predicates, we've already identified some with evaluating operators, for example #15145.

We have some opportunities to statically optimize queries, but I'm going to elide discussion of that.
The main opportunities to optimize queries depend largely on *runtime state.* For example, knowing whether a repository is indexed or not determines whether we can use Zoekt to efficiently compute a result or not. Once we understand the runtime state, we can initiate query planning, which is a fancy way of saying "we translate the initial query at runtime to something a backend understands, making evaluation more efficient". The crux of implementing these optimizations is introducing a translation layer, something like:


Query planning (translation)
Initial user query
Zoekt backend
Searcher backend
Other backends (resolve repos in DB, calculate size, etc.)

Since runtime state can change (say, a repository goes from being unindexed to indexed, or a repository is not indexed yet), the output of query planning will vary in points of time. So it is not an ideal representation to express a search from a user perspective, but obviously useful as a way to define and represent (and reveal to the user) how an initial query was evaluated at runtime.

Our user-facing query language is defined by the following AST (based off of the TypeScript definition):

```
Node = Operator | Parameter | Pattern

/**
 * Terminals
 */
Pattern {
    kind: PatternKind
    value: string
    negated: boolean
}

Parameter {
    field: string
    value: string
    negated: boolean
}

/**
 * Nonterminal for operators AND and OR.
 */
Operator {
    operands: Node[]
    kind: OR | AND
}
```

To define a runtime representation of a query in the above form depends on the components that compute results in the result set definition. These components are at least:

- The PostgreSQL DB (repos)
- Zoekt (repos, files, contents/regex)
- Searcher (repos, files, contents/regex)
- Git (commits, diffs)
- Comby (contents/structural)

And later:

- Codeintel components

And so on.

Expressing a runtime data structure that encapsulates a query's representation comes down to (a) defining data types corresponding to the components above (b) creating an execution path that uses those data types to run those components. The closest semblance of the data type in (a) that we have right now is in internal/search/types.go. The closest semblance of (b) is this switch statement in search_results.go. The biggest limiting factor here is not defining a data type. For example, I can define it in a minute or two, here's a sketch of the definition in pseudocode. This would be the runtime target query of the user's input query:

```
RuntimeQuery = Operator | ComponentQuery
```

```
ComponentQuery = DB | Zoekt | Unindexed | Structural | Git | Lsif | ...

/**
 * Terminal queries for components that produce results
 */
DB {
    query: DBQuery /* native DB query */
}

Zoekt {
    query: ZoektQuery /* native Zoekt query */
}

Unindexed {
    pattern: ... /* this would have the members of, e.g., type TextParameters */
    ...
}

Structural {
    pattern: ...
    ...
}

Git {
   kind: Commit | Message | author | ...
   pattern: ...
   ...
}


/**
 * Nonterminal for runtime query with operators AND and OR. We use a general merge function
 * for results of disjoint kinds of RuntimeQuery.
 */
Operator {
    Query: RuntimeQuery[]
    kind: OR | AND
}
```

Not shown: Every function that takes a `ComponentQuery` as input produces a result in the result set definition.

The biggest limiting factor is that we need to address refactoring around the components that evaluate the kinds of results we need. This has been a long-standing issue (see, e.g., #13319). It helps little if we have a nice way to represent what we want to evaluate, but no way to effectively communicate that representation to the scattered/entangled backend code that is ultimately responsible for running the components above.

For this proposal specifically, I'm hopeful that `contains` have a straightforward translation for Zoekt, and a generic fallback in the absence of Zoekt. I do want these additions to initiate the query planning as part of this proposal. I.e., we can start implementing the `Zoekt` definition and try refactor our code to target that form and evaluate it, since we're most likely to benefit from Zoekt optimizations at this point.

Frontend

Usability in the webapp. We want to enhance discoverability and usability with query suggestions, contextual highlighting, hover information, and diagnostics. This is part and parcel of extending a query language.

Timeline estimate

Rough estimates of how long I think it would take for one engineer to work on these problems. I am nominating myself to take this on when we're ready, and could use an additional engineer to move faster.

- Implementing `select:` in the backend takes **2 weeks**. This mostly because we'll need to restructure surrounding code and do this cleanly. Frontend additions (hovers, diagnostics, validation): **1** week (I want to say 0.5 weeks but I know we have debt to deal with here). Dedicated integration testing: **0.5** weeks. Total: **3.5 weeks**.
- Implementing `contains` in the backend takes **1 week** to add scaffolding for adding general predicate filters (i.e., the Rule Engine code that does processing on search results). Implementing the behavior of contains takes **2 weeks**. Frontend additions (hovers, diagnostics, highlighting, validation) takes **1 week** to add scaffolding for general query support, additional **0.5** weeks to add these details when introducing `contains`. Total: **4.5 weeks**.
- Adding usage metrics for these features takes **1 to 2 weeks** (more difficult to say here, I'm not sure how fluid our current metrics pipeline is). I think metrics are particularly important for these features to surface discoverability and usage. Total: **1 to 2 weeks**.

So by rough estimate, two full-time engineers on this proposal will take about 1.5 iterations to build working functionality for the queries labeled by the 🔨 column. I believe the work seperates cleanly into well-defined tasks, so I assume it can be divided uniformly and proportionally.

Feedback

I'm most interested in initial feedback from people who work at Sourcegraph to validate the direction of this proposal. User testing and hallway tests would be a nice-to-have signal at some point, but not something to kick off with, since there aren't many open questions or concerns at this stage of the proposal. I've toyed with the idea of broadcasting a short version of this proposal and a follow-up RFC on *Code Search Built-ins* to a wider developer audience (i.e., I tweet about it) to gauge interest. We can also open it up to customers as an "FYI, we're embarking on this, what are your thoughts?". I think talking about the evolution of search in a broader sense stimulates interesting ideas and new possibilities, much like some of the issues in this RFCs background (if nothing else) and we should be in the forefront of that category :-).

Definition of success

We implement `select:` and a `contains` filter predicate for `repo:` and `file:` in our query language. We've seen repeated requests for some way to select data results, and I expect we'll see a significant usage uptick once we make it available. There is a discrepancy between the (positive) value and utility we can provide and the (negative) lack of discoverability and usability in our query language. Currently, ad hoc filters `repoHasFile` and `repoHasCommitAfter` see very low usage on existing instances. We know that these constructs have utility and have been requested for compelling use cases in the past. By tracking usage I expect that we'll see significant uptick for the use cases and value delivery of search once select and filter predicates are available.

Addendum: `select:` definitions and behavior

After some back and forth on which values to allow users to specify for `select`, and how it would behave with our current definitions. What prompted this was the process of speccing out select values. At first, the proposal was to define selections with respect to the result set definition, as a path on the tree (which means that the value being selected would always have a unique label). E.g., would we allow expressions like:

`select:repo.file` // give this a meaning like: "select file results, where repo is a qualifier for file"
`select:file` // an alias for `repo.file,` where we don't need to qualify with repo explicitly

This lead to follow-on questions, like whether we would allow selecting on "fields" of current result types, i.e., would we allow to express:

`select:file.repo` // give this a meaning like: "for a file result, select the repo associated with it"

There's clearly some opportunity for confusion based on the order and definition of these values above. After some deliberation, we/I settled on the following, in the interest of: (a) make usage intuitive (b) don't close off opportunities to extend `select` for broader use cases.

In the interest of (a), we threw out the idea of representing select values with full qualifiers like `a.b.c`. Instead, the basics of select will simply implement selection on data for 5 basic kinds:

- `repo`
- `file`
- `content`
- `commit`
- `symbol`

The behavior of select, when specifying these 5 basic kinds, is defined in the below pseudo code.

In the interest of (b), we extend the possibility of values with one or more `.` (dot access), to allow notions like `select:symbol.variable`. Here, a `.` access means filter symbol results such that the symbol kind is `variable`.

Specifying a kind implies that the definition of our 5 basic kinds above has such a (sub)kind. We do not necessarily define `.` access on all subfields--we are just saying that if we define selecting on something like `select:symbol.variable`, then `variable` must exist as a kind on the definition of `symbol` so that we can compare values and select those that equal the specified kind.

This does not close off the opportunity to extend our 5 basic kinds or subkinds in future (e.g., for possibilities like `select:issue.comment` versus `select:issue.description`). It’s possible to nest subkinds--demonstrating one level is enough to show the convention generalizes.

The pseudocode below lays out a model of the type definitions and semantics. There is more than one way to represent this, but at least one way is sufficient :-)

The ideal type definitions for results in part `(** (1) Data type definitions *)`. They roughly correspond to our top-level GQL definition of SearchResult, but our GQL definition is underspecced for these disjoint results, IMO. Our top-level GQL definition is:

```
union SearchResult = FileMatch | CommitSearchResult | Repository
```

The pseudocode lays out the semantics of select in part `(** (2) Select semantics  *)`, which corresponds to our implementation in Sourcegraph.

```
(** (1) Data type definitions *)
type repo =
  { name : string
  }

type file =
  { repo : repo
  ; name : string (* or path name *)
  }

type content =
  { repo : repo
  ; file : file
  ; matches : string list
  }

type symbol =
  { repo : repo
  ; file : file
  ; kind : string (* e.g., "module", "variable", "constant" *)
  ; matches : string list
  }

(* just an example of commit data *)
type commit_data =
  | Diff of string
  | Message of string
```

```
type commit =
  { repo : repo
  ; files : file list
  ; matches : commit_data
  }

type result =
  | Repo of repo
  | File of file
  | Content of content (* e.g., line match, to distinguish from symbols *)
  | Symbol of symbol
  | Commit of commit


(** (2) Select semantics *)

(**
    [mapper] is a function that maps any of our results to a list of values.
    Specific functions implement logic to map a result to, e.g., the [repo] or
    [file] values above. Internally, mapper functions are available as a
    convenience function for extracting data from [result] types (e.g., [repo],
    [file]). The workhorse function [select_basic] uses convenience mappers and
    then constructs a list of [result] types from the list of values based on the
    selector.

    I.e., we could easily alternatively define the mapper as:

    type mapper = result -> result

    with some corresponding implementation details to, e.g., [to_repo] or
    [to_file] mapper functions below, except that it's convenient to create
    mapper functions that can extract members of [result] for internal use.
*)
type 'a mapper = result -> 'a list

(* select:repo *)
let to_repo : repo mapper =
  fun (result : result) : repo list ->
  match result with
  | Repo repo -> [repo]
  | File { repo; _ } -> [repo]
  | Content { repo; _ } -> [repo]
  | Symbol { repo; _ } -> [repo]
  | Commit { repo; _ } -> [repo]

(* select:file *)
let to_file : file mapper =
  fun (result : result) : file list ->
  match result with
  | Repo _ -> []
  | File file -> [file]
  | Content { file; _ } -> [file]
  | Symbol { file; _ } -> [file]
  | Commit { files; _ } -> files

(* select:symbol.kind *)
let to_symbol_kind : string -> symbol mapper =
  fun (select_kind : string) (result : result) : symbol list ->
  match result with
  | Symbol ({ kind; _ } as symbol) when String.(kind = select_kind) -> [symbol]
  | Symbol _
  | Repo _
  | File _
  | Content _
  | Commit _ -> []

(* Basic select:value syntax *)
let select_basic (selector : string) results : result list =
  match selector with
  | "repo" -> List.concat_map to_repo results |> List.map (fun repo -> Repo repo)
  | "file" -> List.concat_map to_file results |> List.map (fun file -> File file)
  | "content" -> failwith "not implemented, you get the idea"
  | "symbol" -> failwith "not implemented, you get the idea"
  | "commit" -> failwith "not implemented, you get the idea"
  | _ -> failwith "selector not supported"

(**
    Extended select:value.kind syntax. Can nest, e.g., select:value.k1.k2. This
    demonstrates one level, details for further nestedness elided for brevity
*)
let select_kind (selector : string) (kind : string) results : result list =
  match (selector, kind) with
  | "symbol", kind ->
     List.concat_map (to_symbol_kind kind) results
     |> List.map (fun symbol -> Symbol symbol)
  | _ -> failwith "selector and kind not supported"

(**
   Two kinds of select syntax:

   select:value
   select:value.kind
*)
let select (query : string) results : result list =
  let path = String.split_on_char '.' query in
  match path with
  | [] -> results
  | selector :: [] -> select_basic selector results
  | selector :: kind :: [] -> select_kind selector kind results
  | _ -> failwith "selector not supported"
```