Glean

FACTS ABOUT CODE

Simon Marlow

Meta

“I want to ask questions about my code”

What are all the functions in file `xyz.hs`?

Where's the definition of `my_function`?

Who is using `MyClass`?

What are all the functions in file `xyz.hs`?

Where's the definition of `my_function`?

Who is using `MyClass`?

With these, you could implement IDE features like jump-to-definition, find-all-references, or outlines.

What are all the headers referenced by my project?

Is anything from this import used in this file?

Who is calling `my_function`?

What are all the headers referenced by my project?

Is anything from this import used in this file?

Who is calling `my_function`?

With these, you could implement dead code detection tools.

What are all the methods of C and their types?

What does the class C inherit from?

What documentation comments are associated with C?

What are all the methods of C and their types?

What does the class C inherit from?

What documentation comments are associated with C?

With these, you could implement code documentation tools.

Find all functions called `unsafeXXX`

Find all the functions that return a T

Find code that creates an object of type MyClass

Find all functions called `unsafeXXX`

Find all the functions that return a T

Find calls to `my_function`

With these, you could implement code search tools.

All of these things exist

- In one form or another

All of these things exist

- In one form or another
- But what if you want to ask questions about
  - All of your company's code?
  - Or all of github?
  - In all the different programming languages?

All of these things exist

- In one form or another
- But what if you want to ask questions about
  - All of your company’s code?
  - Or all of github?
  - In all the different programming languages?

We need something that scales

- to large codebases
- to multiple programming languages
- to lots of clients
- to complex queries

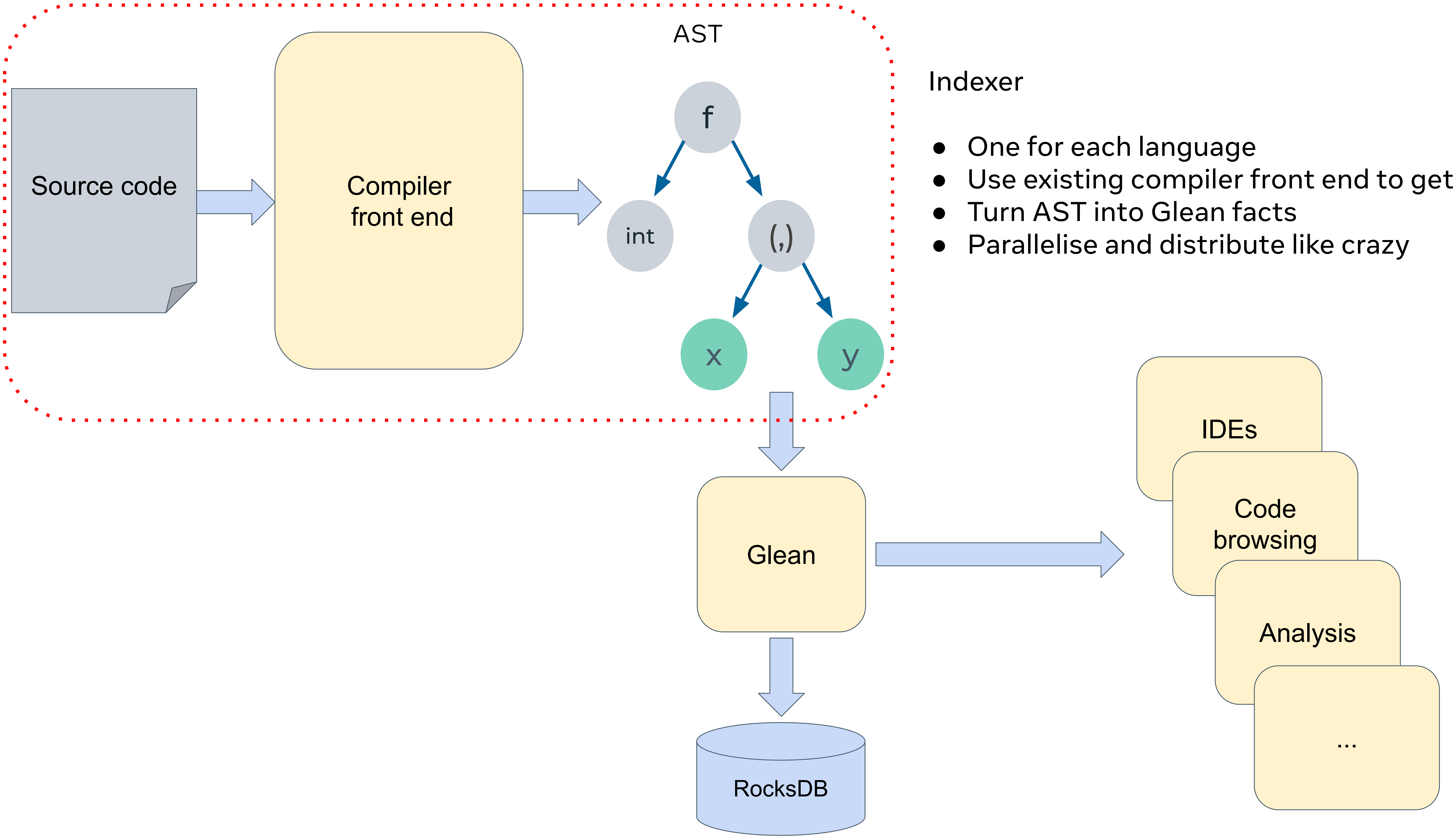
Glean

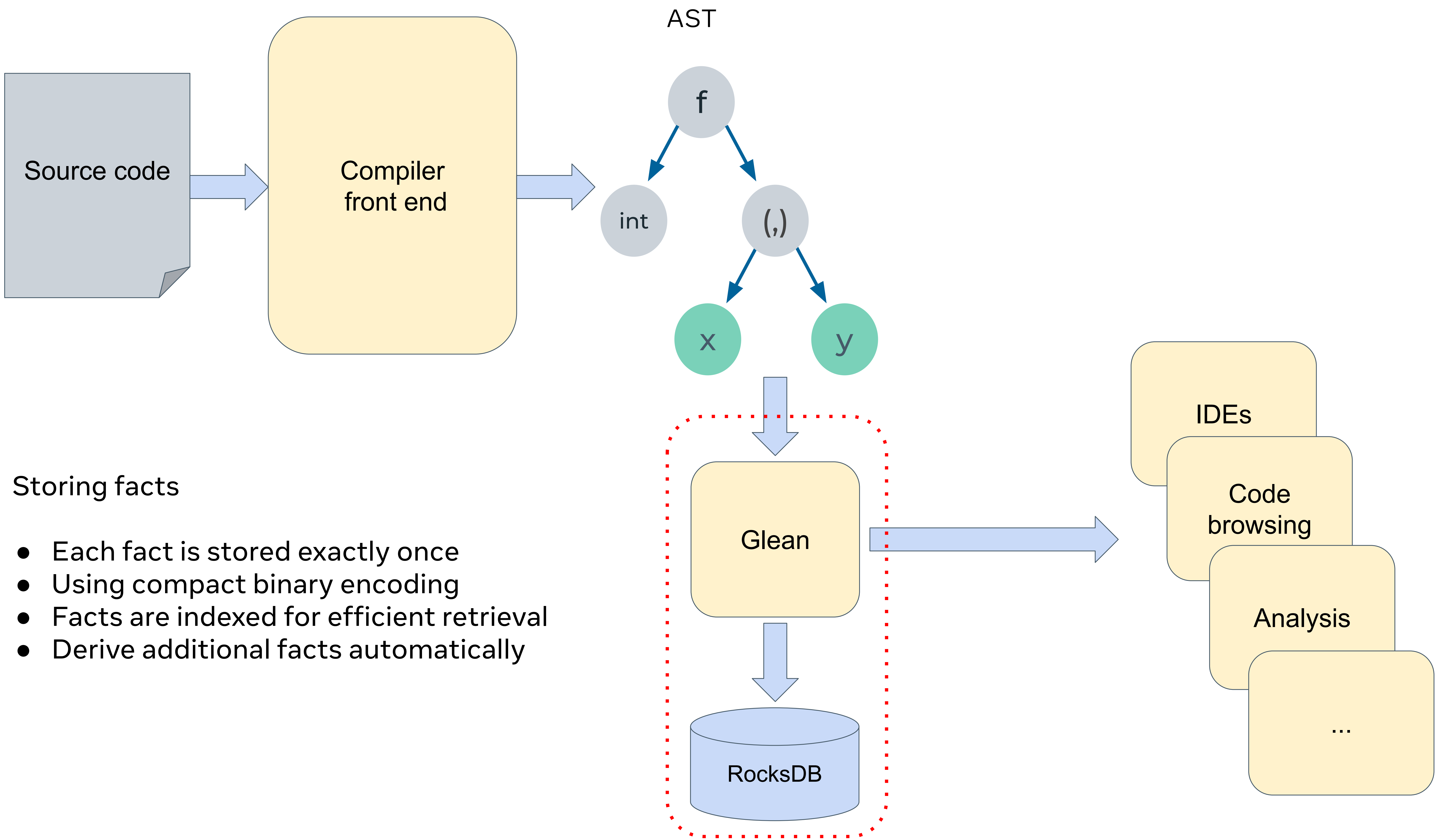
System for collecting, deriving and querying facts about source code

Get Started

Indexer

- One for each language
- Use existing compiler front end to get AST
- Turn AST into Glean facts
- Parallelise and distribute like crazy



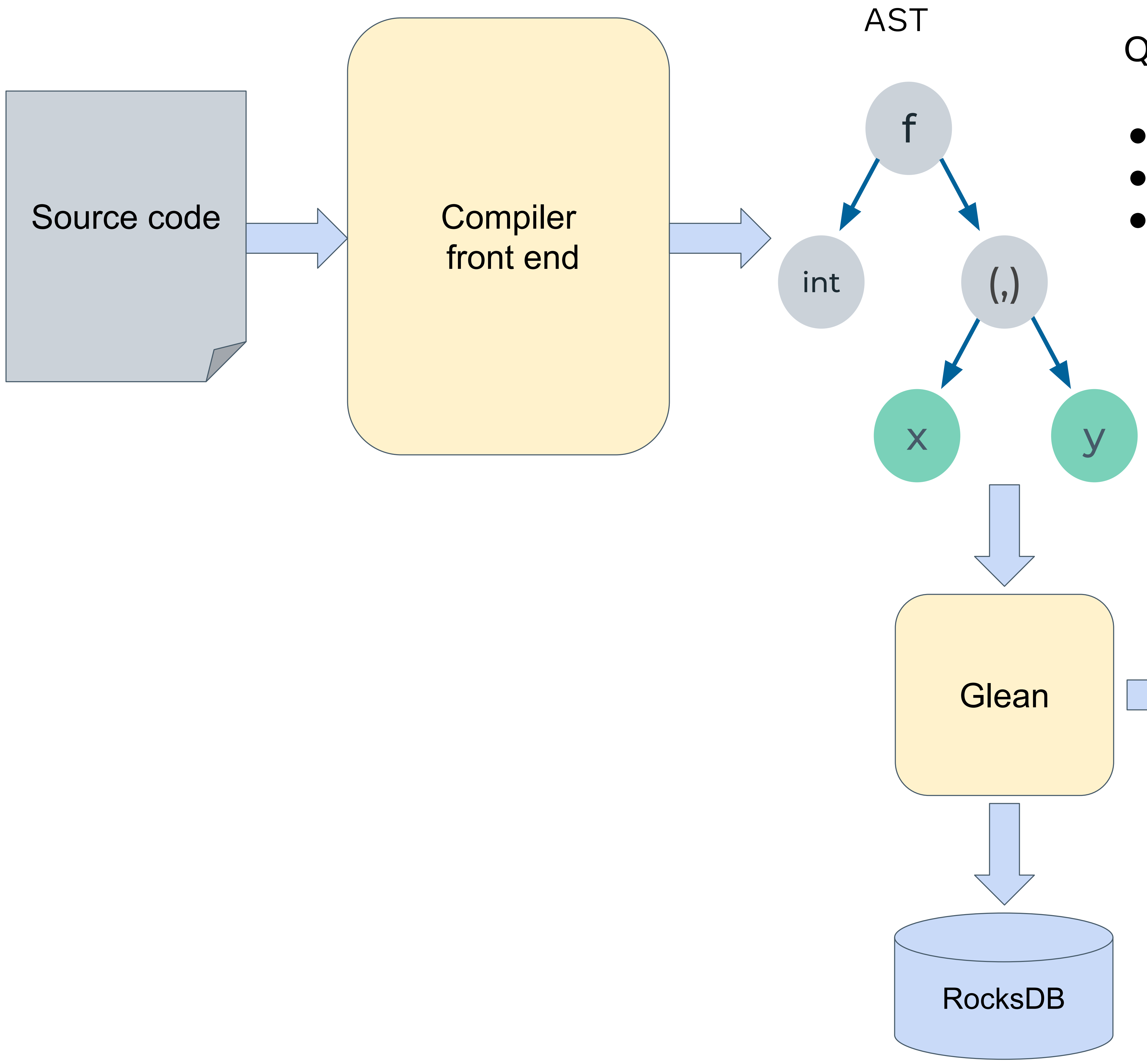


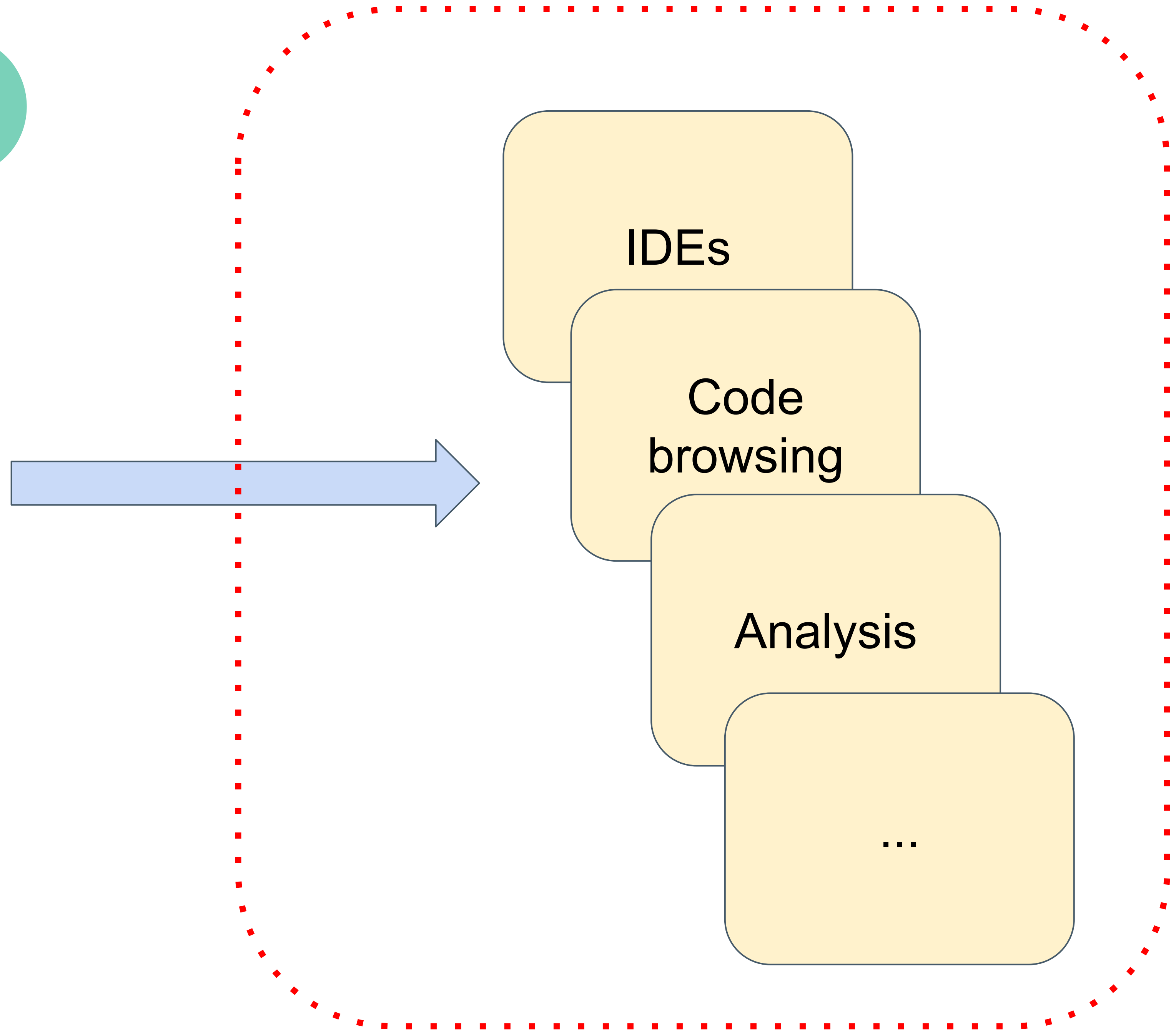
Storing facts

- Each fact is stored exactly once
- Using compact binary encoding
- Facts are indexed for efficient retrieval
- Derive additional facts automatically

Querying the data

- Glean’s query language is called ***Angle***
- Based on Datalog
- Querying at scale:
  - DBs replicated across a fleet of servers
  - Indexing and replication runs continuously





Basic principles

Glean stores *facts*

Each fact is stored exactly once, and is given a unique *fact ID*

The type of a fact is a *predicate*

Predicate	Fact
`predicate Name: string`	`{ “i`

This is what the programmer writes in the **schema**.

It defines the shape of the data to store.

This says we have a predicate “Name” with keys of type “string”

Predicate	Fact
`predicate Name: string`	`{ "id": 42, "key": "foo" }`

This is an example of a Name fact.

I'm using JSON as a concrete representation, but you can also write and query data using a strongly-typed DSL.

The ID 42 is assigned by Glean; if you write the same fact twice they get commoned up.

Predicate	Fact
`predicate Name: string`	`{ “id”: 42, “key”: “foo” }`

`predicate QualifiedName:`
`  {`
`    name : Name,`
`    module : Name`
`  }`
	`{`
	`  “id”: 43,`
	`  “key”: {`
	`    “name”: { “id”: 44, “key”: “f” }`
	`, “key”, “M” }`

Predicates can refer to other predicates, so the facts form a DAG. (note *acyclic)*

Predicate	Fact
predicate Name: string	{ “id”: 42, “key”: “foo” }

predicate QualifiedName:
  {
    name : Name,
    module : Name
  }
	{
	  “id”: 43,
	  “key”: {
	    “name”: { “id”: 44, “key”: “f” }
	    “module”: { “id”: 45, “key”, “M” }
	  }
	}

You can write data nested like this.

Query results are also expanded by default.

<table>
<tr><th>Predicate</th><th>Fact</th></tr>
<tr><td><pre>predicate Name: string</pre></td><td><pre>{ “id”: 42, “key”: “foo” }</pre></td></tr>
<tr><td><pre>predicate QualifiedName:
  {
    name : Name,
    module : Name
  }</pre></td><td><pre>{
  “id”: 43,
  “key”: {
    “name”: { “id”: 44, “key”: “f” }
    “module”: { “id”: 45, “key”, “M” }
  }
}</pre></td></tr>
<tr><td><pre>type Location =
  {
    line : nat,
    column : nat
  }

predicate Class :
  {
    name : QualifiedName,
    location : Location,
  }</pre></td><td><pre>   }

      “column”: 0
    }
  }
}</pre></td></tr>
</table>

This is a type, not a predicate. It has no facts, it’s just an alias.

Predicate	Fact
`predicate Name: string`	`{ “id”: 42, “key”: “foo” }`

`predicate QualifiedName:`
`  {`
`    name : Name,`
`    module : Name`
`  }`
	`{`
	`  “id”: 43,`
	`  “key”: {`
	`    “name”: { “id”: 44, “key”: “f” }`
	`    “module”: { “id”: 45, “key”, “M” }`
	`  }`
	`}`

`type Location =`
`  {`
`    line : nat,`
`    column : nat`
`  }`
 	
`predicate Class :`
`  {`
`    name : QualifiedName,`
`    location : Location,`
`  }`
	`{`
	`  “id”: 44,`
	`  “key”: {`
	`    “name” : { “id”: 43 }`
	`    “Location” : {`
	`      “line”: 10,`
	`      “column”: 0`
	`    }`
	`  }`
	`}`

We might refer to a fact by its ID only, to express sharing explicitly.

Indexing end-to-end

- Define a schema (predicates + types)
- The indexer emits data using either JSON or a typed DSL
- Glean encodes and writes the data to a RocksDB
- Then you can query it…

Querying with Angle

```
$ glean shell
Glean Shell, built on 2022-04-27T14:22:02Z, from rev
ed19a6d3e3c3a66867a16f988a134f7a9b7f40d3
type :help for help.
>
```

Querying with Angle

```
> example.Class { name = { name = "MyClass" }}
```

This is a query, it will return all matching facts in the DB

Querying with Angle

```
> example.Class { name = { name = "MyClass" }}
```

The query starts with a predicate

Querying with Angle

```
> example.Class { name = { name = "MyClass" }}
```

and gives a pattern to specify
which facts we want to match

Querying with Angle

```
> example.Class { name = { name = "MyClass" }}
```

and gives a pattern to specify
which facts we want to match

```
predicate Name: string
```

```
predicate QualifiedName:
  {
    name : Name,
    module : Name
  }
```

```
predicate Class :
  {
    name : QualifiedName,
    location : Location,
  }
```

Querying with Angle

```
facts> example.Class { name = { name = "MyClass" }}
{
  "id": 1027,
  "key": {
   "name": {
   "id": 1026,
   "key": {
      "name": { "id": 1024, "key": "MyClass" },
      "module": { "id": 1025, "key": "M" }
   }
   },
   "location": { "line": 10, "column": 1 }
  }
}

1 results, 4 facts, 0.57ms, 215504 bytes, 1313 compiled bytes
facts>
```

Querying with Angle

- The query language is quite expressive:
  - statements, variables, disjunction, if-then-else, negation
  - supported by a query compiler, optimizer, and byte-code query engin

Querying with Angle

- The query language is quite expressive:
  - statements, variables, disjunction, if-then-else, negation
  - supported by a query compiler, optimizer, and byte-code query engin

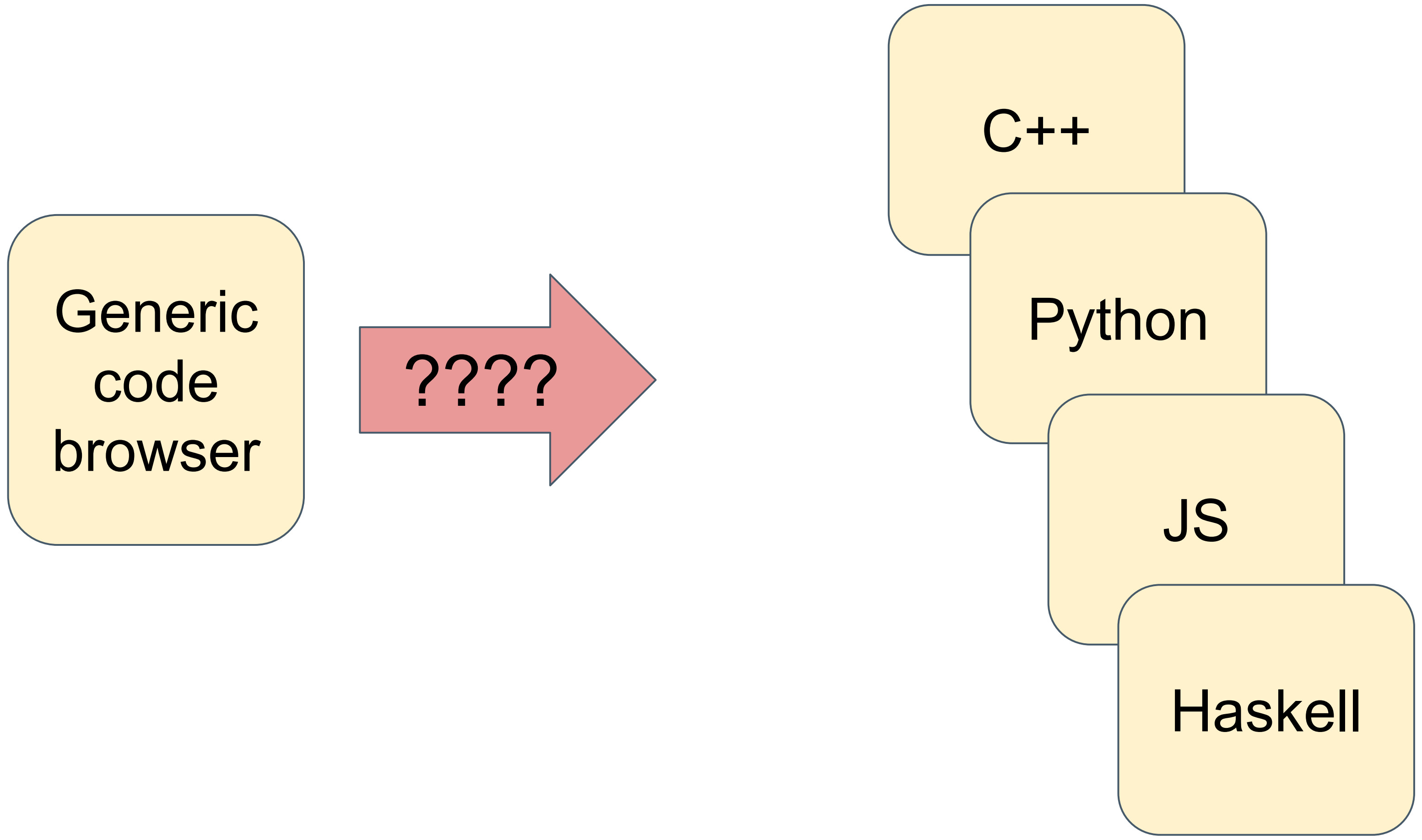
- Recursive queries? not yet.

Philosophy: language-specific schemas

- Each language has its own schema
- Language-specific detail enables language-specific tools
  - e.g. redundant `#include` removal in C++
- Design the schema to be a natural fit for the indexer

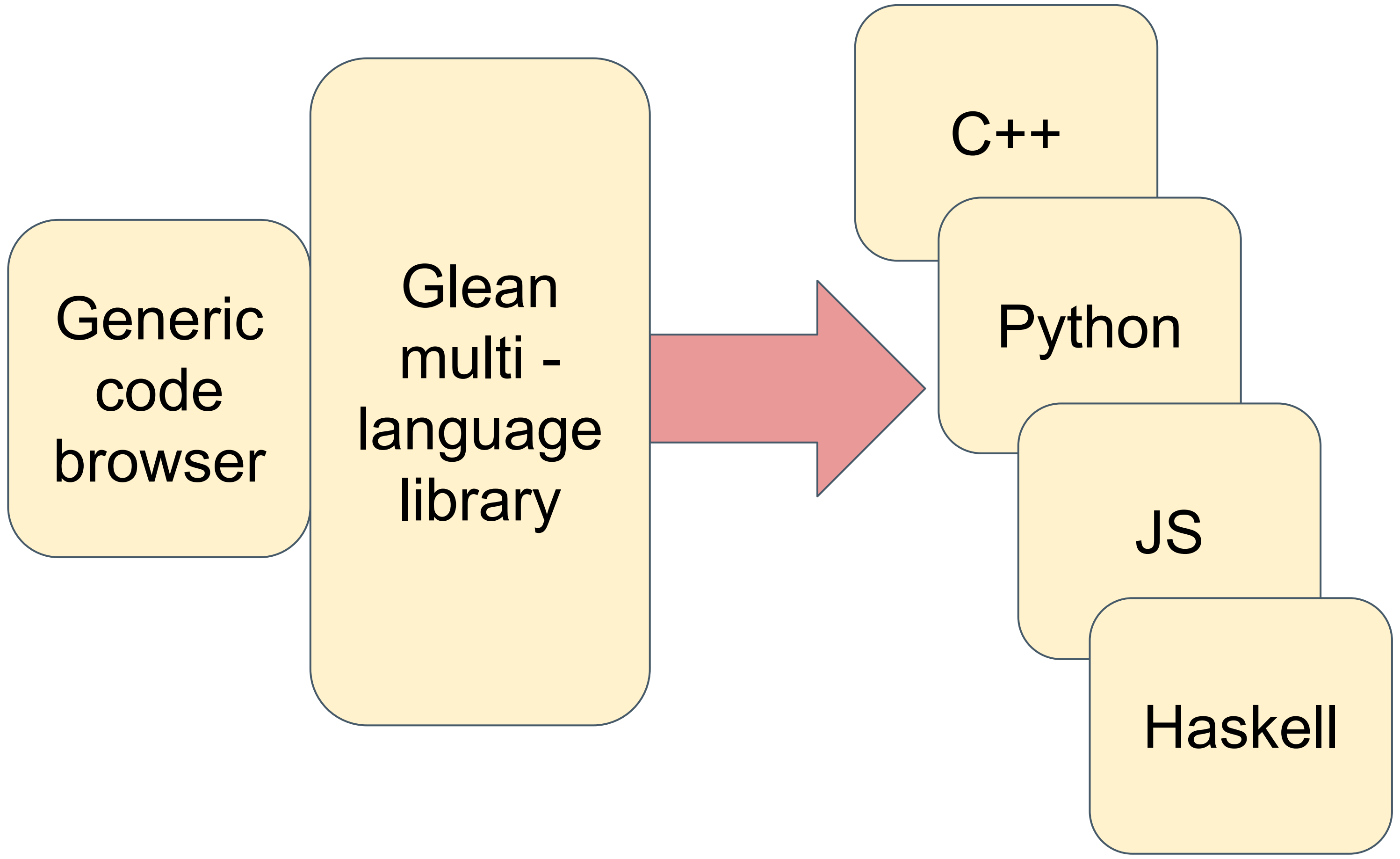
But how will we support language-agnostic tools?

- With no commonality across our language-specific schemas, it will be hard for clients to use the data



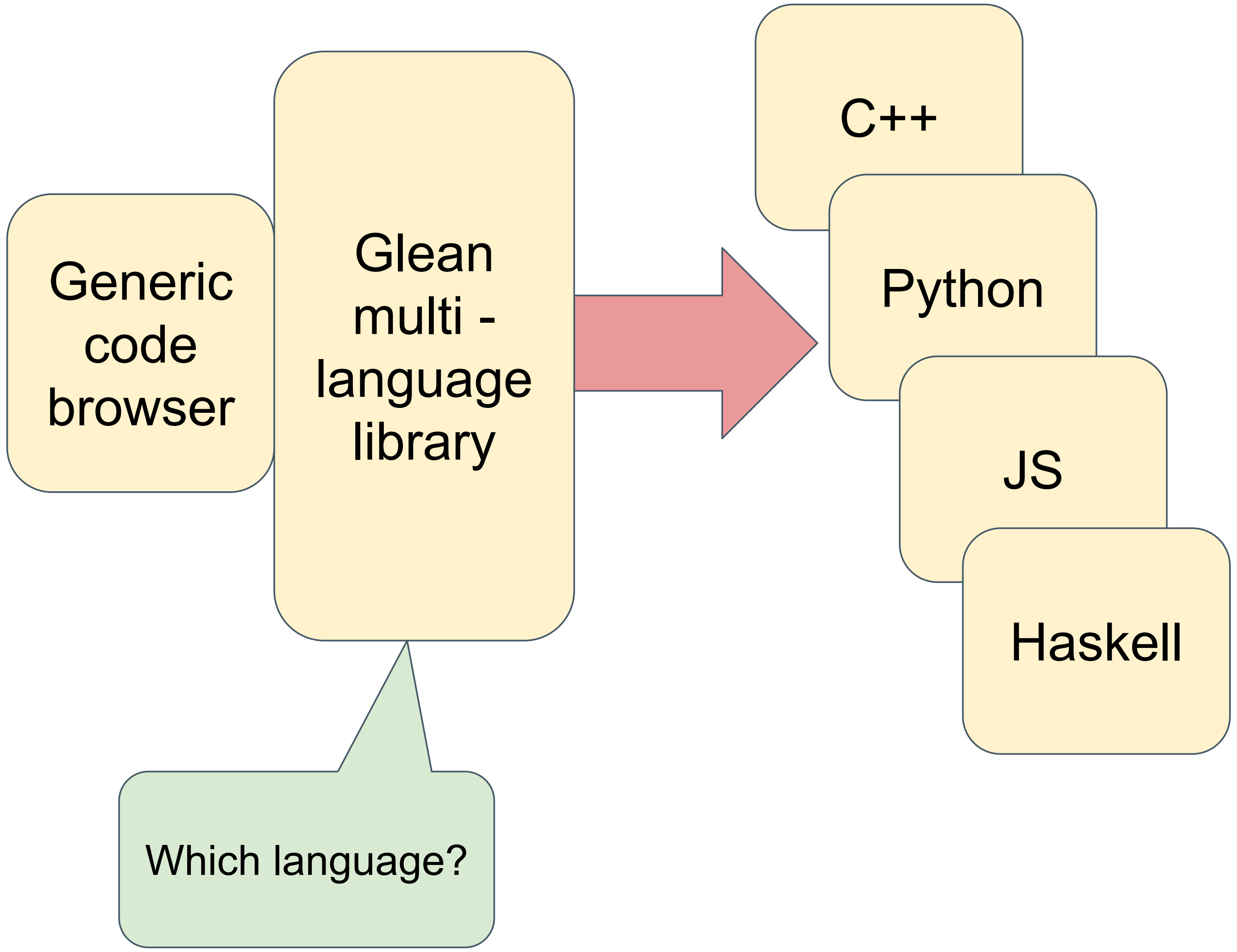
But how will we support language-agnostic tools?

- Build a client-side library to abstract over the data?



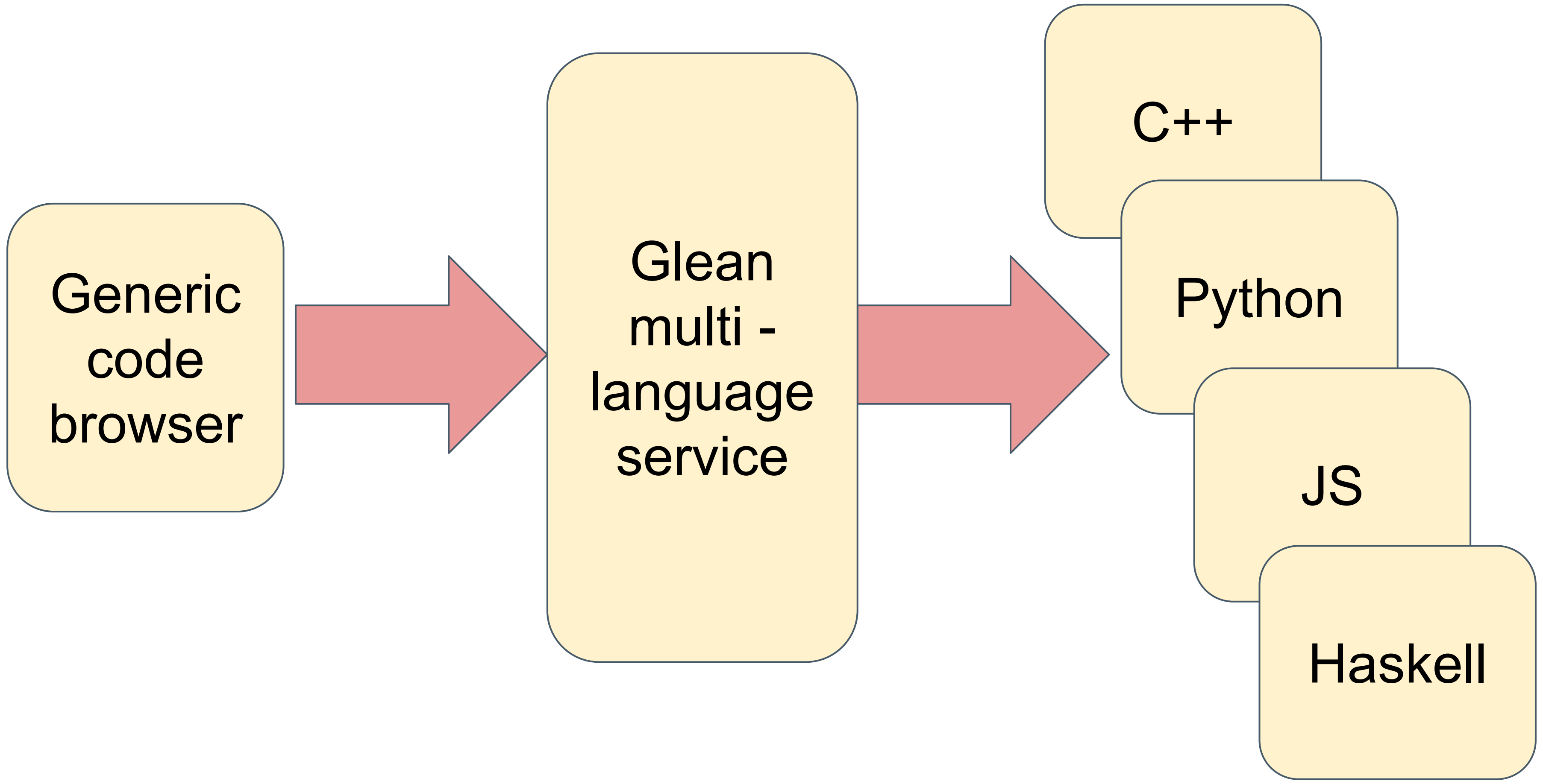
But how will we support language-agnostic tools?

- Build a client-side library to abstract over the data?
  - But we want to support clients in multiple languages, so clients must be thin



But how will we support language-agnostic tools?

- Put it in a service?
- Yes, but the API is fixed and inflexible, can’t write arbitrary queries



Datalog

- Datalog is a query language in which you can *derive* new facts from existing facts
- e.g. we have
  - Facts about Java declarations in Java source files
  - Facts about C++ declarations in C++ source files
  - …
- From all these we can derive:
  - *Facts about **declarations** in **source files***

“codemarkup”

```
type Declaration =
  {
    java : java.Declaration |
    python : python.Declaration |
    ...
  }

predicate FileDeclaration :
  {
    file : string,
    decl : Declaration
  }
  { File, Decl } where
    JavaFileDeclaration { File, Decl } |
    PythonFileDeclaration { File, Decl } |
    ...
```

Declaration: a declaration in any language

“codemarkup”

```
type Declaration =
  {
    java : java.Declaration |
    python : python.Declaration |
    ...
  }

predicate FileDeclaration :
  {
    file : string,
    decl : Declaration
  }
  { File, Decl } where
    JavaFileDeclaration { File, Decl } |
    PythonFileDeclaration { File, Decl } |
    ...
```

FileDeclaration { F, D }
“file F contains declaration D”

“codemarkup”

```
type Declaration =
  {
    java : java.Declaration |
    python : python.Declaration |
    ...
  }

predicate FileDeclaration :
  {
    file : string,
    decl : Declaration
  }
  { File, Decl } where
    JavaFileDeclaration { File, Decl } |
    PythonFileDeclaration { File, Decl } |
    ...
```

Defined as the disjunction of all language-specific FileDeclaration predicates

“codemarkup”

```
type Declaration =
  {
    java : java.Declaration |
    python : python.Declaration |
    ...
  }

predicate FileDeclaration :
  {
    file : string,
    decl : Declaration
  }
  { File, Decl } where
    JavaFileDeclaration { File, Decl } |
    PythonFileDeclaration { File, Decl } |
    ...
```

Queries

```
FileDeclaration { file = “foo.py” }
```

Would search for declarations of all languages in “foo.py” and return them

Glean’s “codemarkup” layer

- We’ve built a library of language-agnostic predicates called “codemarkup”
- Currently supports common code navigation properties:
  - Declarations-by-file
  - References-by-file
  - Declaration-to-uses
  - Some declaration metadata
- Languages:
  - C++, Python, Javascript (Flow), C++, Objective C, Haskell, Rust, Erlang, generic LSIF

Real-world example

```
$ glean index flow ~/code/react --repo react/1
…
```

Real-world example

```
$ glean index flow ~/code/react --repo react/1
... (a few seconds later)
Wrote facts about 616 JavaScript files.
...
$
```

Real-world example

```
$ glean shell
Glean Shell, built on 2022-04-22 09:45:37.840354178 UTC, from rev
cf3b295281c94578945c5010b20cc1bad2e81a7f
type :help for help.
>
```

Real-world example

```
$ glean shell
Glean Shell, built on 2022-04-22 09:45:37.840354178 UTC, from rev
cf3b295281c94578945c5010b20cc1bad2e81a7f
type :help for help.
> :db react/1
react>
```

Real-world example

```
$ glean shell
Glean Shell, built on 2022-04-22 09:45:37.840354178 UTC, from rev
cf3b295281c94578945c5010b20cc1bad2e81a7f
type :help for help.
> :db react/1
react> :stat
...
Total: 431690 facts (15.37 MB)
```

Real-world example

```
react> codemarkup.FileEntityLocations { file = "test/packages/shared/ReactTypes.js" }
{
  "id": 432714,
  "key": {
   "file": { "id": 90012, "key": "test/packages/shared/ReactTypes.js" },
   "location": {
   "name": "ReactScopeInstance",
   "file": { "id": 90012, "key": "test/packages/shared/ReactTypes.js" },
   "location": { "span": { "start": 1927, "length": 18 } }
   },
   ...
}

1 results, 6 facts, 15.77ms, 13167976 bytes, 64383 compiled bytes
results truncated (current limit 1, use :limit <n> to change it)
Use :more to see more results
react>
```

What about functional programming???

With Haskell we can define a type-safe query DSL

Angle	Haskell
`src.File “foo.hs”`	`predicate @Src.File $ string “foo.hs”`

Type application tells the DSL which predicate we’re searching

```
predicate :: forall p . Predicate p => Angle (KeyType p) -> Angle p
string :: Text -> Angle Text
```

With Haskell we can define a type-safe query DSL

Angle	Haskell
`src.File “foo.hs”`	`predicate @Src.File $ string “foo.hs”`

“Angle t” is a query that returns results of type t.

It pretty-prints as the actual query.

```
predicate :: forall p . Predicate p => Angle (KeyType p) -> Angle p
string :: Text -> Angle Text
```

With Haskell we can define a type-safe query DSL

<table>
<tr><th>Angle</th><th>Haskell</th></tr>
<tr><td><pre>src.File “foo.hs”</pre></td><td><pre>predicate @Src.File $ string “foo.hs”</pre></td></tr>
<tr><td><pre>python.DeclarationWithName {
  name = “foo”
}</pre></td><td><pre>predicate @Python.DeclarationWithName $
  rec $
    field @”name” “foo”
  end</pre></td></tr>
</table>

Type application with string type literal for field names.

With Haskell we can define a type-safe query DSL

Angle	Haskell
`src.File “foo.hs”`	`predicate @Src.File $ string “foo.hs”`

`python.DeclarationWithName {`
`  name = “foo”`
`}`
	`predicate @Python.DeclarationWithName $`
	`  rec $`
	`    field @”name” “foo”`
	`  end`

The DSL has enough information to type-check the record field.

Instant feedback in VS code

```
• No instance for (HasField "nosuchfield" Text 'TNoFields)
    arising from a use of ‘rec’
• In the second argument of ‘predicate’, namely
    ‘(rec
        $ field @"file" (asPredicate (factId fileid))
            $ field @"location" location
                $ field @"entity" entity
                    $ field @"nosuchfield" (string "abc") end)’
  In the second argument of ‘(.=)’, namely
    ‘predicate
       @Code.FileEntityLocations
       (rec
          $ field @"file" (asPredicate (factId fileid))
          field @"nosuchfield" (string "abc")
        end)
      ]
```

Instant feedback in VS code

```
--
• Couldn't match type ‘Glean.Nat’ with ‘Code.Location’
    arising from a use of ‘rec’
• In the second argument of ‘predicate’, namely
    ‘(rec
        $ field @"file" (asPredicate (factId fileid))
            $ field @"location" (nat 42) $ field @"entity" entity $ end)’
  In the second argument of ‘(.=)’, namely
    ‘predicate
       @Code.FileEntityLocations
       (rec
          $ field @"file" (asPredicate (factId fileid))
              $ field @"location" (nat 42) $ field @"entity" entity $ end)’
  In the expression:
          field @"location" (nat 42) $
          field @"entity" entity $
        end)
      ]
```

How does this work?

schema.angle

```
predicate FileEntityLocations:
  {
   file: src.File,
   location: Location,
   entity: code.Entity,
  }
```

schema.hs

```
data FileEntityLocations = FileEntityLocations
  { fileEntityLocations_file :: Src.File
  , fileEntityLocations_location :: Code.Location
  , fileEntityLocations_entity :: Code.Entity
  }

type instance RecordFields FileEntityLocations =
  TField "file" Src.File (
  TField “location” Code.Location (
  TField “entity” Code.Entity
  TNoFields))
```

Code generation from the schema to Haskell (amongst other languages)

How does this work?

schema.angle

```
predicate FileEntityLocations:
  {
   file: src.File,
   location: Location,
   entity: code.Entity,
  }
```

schema.hs

```
data FileEntityLocations = FileEntityLocations
  { fileEntityLocations_file :: Src.File
  , fileEntityLocations_location :: Code.Location
  , fileEntityLocations_entity :: Code.Entity
  }

type instance RecordFields FileEntityLocations =
  TField "file" Src.File (
  TField “location” Code.Location (
  TField “entity” Code.Entity
  TNoFields))
```

A generated Haskell datatype for each predicate and type in the schema.

How does this work?

schema.angle

```
predicate FileEntityLocations:
  {
   file: src.File,
   location: Location,
   entity: code.Entity,
  }
```

schema.hs

```
data FileEntityLocations = FileEntityLocations
  { fileEntityLocations_file :: Src.File
  , fileEntityLocations_location :: Code.Location
  , fileEntityLocations_entity :: Code.Entity
  }

type instance RecordFields FileEntityLocations =
  TField "file" Src.File (
  TField “location” Code.Location (
  TField “entity” Code.Entity
  TNoFields))
```

Type instances tell the query library about the fields and their types

A pleasant query authoring experience

- Iterate on your Angle code using Glean to typecheck it
- Generate the Haskell
- Iterate on your Haskell query code, using GHC to typecheck it in VS Code
- … queries do not fail at runtime
- … queries return native Haskell types

A pleasant query authoring experience

- This isn't Haskell-specific.
- There's also a Hack-based DSL for queries
  - it looks very different, but it's idiomatic Hack
- And we're planning a Python DSL too

- Clients using languages without a DSL can make raw Angle queries, just without compile-time typechecking and IDE feedback.

More Haskell benefits: Effortless concurrency with

Haxl

- Clients are often making multiple queries to Glean
- We want those to be concurrent when possible
- Haxl + ApplicativeDo is great for this

ApplicativeDo

Turns

```
do
  a <- x
  b <- y
  return (a,b)
```

into

```
(,) <$> x <*> y
```

by analysing dependencies between statements.

Haxl

Performs Applicatives in parallel.

```
(,) <$> x <*> y
```

will run x and y in parallel.

- Requires a Haxl “datasource” to be implemented for each backend
- Glean includes a Haxl datasource for Glean queries

Haxl

When you write

```
mapM query list
```

the queries all run in parallel.

Haxl

Haxl + ApplicativeDo

When you write

```
do
  a <- .. glean query ..
  b <- .. glean query ..
  return (a,b)
```

with ApplicativeDo enabled, the queries run in parallel.

The Haxl library: effortless concurrency

- A snippet from the Glass codebase that fetches the symbols for a file:

```
{-# LANGUAGE ApplicativeDo #-}

documentSymbolsForLanguage mlimit _ includeRefs fileId = do
  xrefs <- if includeRefs
    then searchRecursiveWithLimit mlimit $
      Query.fileEntityXRefLocations fileId
    else return []
  defns <- searchRecursiveWithLimit mlimit $
    Query.fileEntityLocations fileId
  return (xrefs,defns)
```

These two queries run concurrently!

FAQ

- What languages do you support?
  - Open source now: Javascript/Flow, Hack, Typescript*, Rust*, Go*

    * = via LSIF

  - Open source but not fully integrated: C++ & Objective C, Rust, Haskell
  - Not open source yet, but planned: Python, Java

FAQ

- Are there any actual clients I can use?

FAQ

- Where are you going with open source?

How do I play with it?

- http://glean.software
- Demo Docker images available for download

Meta