Indexing Common Lisp With Kythe

Jonathan Godbout

For ELS 2020

Agenda

- Introduction
- Motivation
- Overview of Kythe
- Output and Tools
- Challenges
- Future Work

About Me

Software Engineer at Google working on QPX

Maintainer of Lisp-Koans:
https://github.com/google/lisp-koans

Writer of Blog:

https://experimentalprogramming.wordpress.com/

PhD Candidate at University of New Hampshire (Mathematics)

TLDR

Kythe is a pluggable system for creating annotated code graphs.

We have developed a Common Lisp (SBCL) indexer plug-in for Kythe.

This will allow you to create UI’s with jump to definition.

We are working on open sourcing the plugin.

You may stop listening… Or I have 25 minutes so…

Motivation

- Code distributed across a code-base is hard to navigate.
- Take the function “verbose” from
  - https://github.com/qitab/bazelisp/blob/master/bazel/log.lisp
- Locally in your file system you have to grep, but this only works for the files you have locally.
- With Slime you still have to load all possible files and then M-. and that misses quite a few cross references.

```
;;;
;;; Simple logging utilities for the Bazel Lisp Compile utility.
;;;

(defpackage bazel.log
  (:use :cl)
  (:export #:info #:message
           #:verbose #:vv #:vvv
           #:fatal #:fatal-error #:non-fatal-error
           #:*verbose*))

(in-package #:bazel.log)

(declaim (fixnum *verbose*))
(defvar *verbose* 0)

(define-condition fatal-error (error)
  ((message :reader fatal-error-message :initarg :message :type (or null string)))
  (:documentation "An error caused by the log:fatal.")
  (:report (lambda (e s)
             (format s "~@[~A~]" (fatal-error-message e)))))

(deftype non-fatal-error () '(and error (not fatal-error)))

(defun message (severity level control &rest args)
  "Format and print a log message.
 The first argument is SEVERITY: :INFO, :WARNING, :ERROR.
 The second argument specifies log verbosity LEVEL.
 The ARGS are applied to the CONTROL string to produce the log output."
  (declare (fixnum level))
  (when (>= *verbose* level)
    (with-standard-io-syntax
      (let ((*print-readably* nil)
            (out (if (eq severity :info)
                     *standard-output*
                     *error-output*)))
        (format out "~&~A: ~?~%" severity control args)))))

(defun verbose (control &rest args)
  "Same as message with level 1. CONTROL is the format control string that operates on ARGS."
  (apply #'message :info 1 control args))
```

And then we don’t even get the right results...

- On Github we can’t find where it’s used without a textual search.
  - I can’t even provide a link to the function itself.
  - I can provide a link to the line number, but that may change.

6 code results in qitab/bazelisp or view all results on GitHub

bazel/log.lisp

```
5   (defpackage bazel.log
6     (:use :cl)
7     (:export #:info #:message
8              #:verbose #:vv #:vvv
9              #:fatal #:fatal-error #:non-fatal-error
10             #:*verbose*))
11
12  (in-package #:bazel.log)
13
14  (declaim (fixnum *verbose*))
```

Common Lisp Showing the top three matches Last indexed on Jul 2, 2018

bazel/main.lisp

```
191     "Add a WARNING to the failures list of the ACTION."
192     (verbose "Added failure: ~S '~A'" (type-of warning) warning)
…
271     (declare (string warning-file) (list warnings))
272     (verbose "Saving ~A warning~:P: ~S" (length warnings) warning-file)
```

Common Lisp Showing the top two matches Last indexed on Jul 2, 2018

bazel/rules.bzl

```
100           cfg="host",
101           default=Label(BAZEL_LISP)),
102       "verbose": attr.int(),
103       # Internal, for testing coverage.
…
169   def _default_flags(ctx, trans, verbose_level):
170     """Returns a list of default flags based on the context and trans provider.
```

Python Showing the top two matches Last indexed on Jul 2, 2018

Our tools should help us...

- We should be able to click on verbose and get every reference.
- We should be able right click on verbose and get a link directly to verbose.
- We should not get Python file references unless they are intra-language function calls.
- We should not have to compile all possible lisp code into a REPL.

What is Kythe?

- From https://kythe.io/:
  - *“A pluggable, (mostly) language-agnostic ecosystem for building tools that work with code.”*
- What does that mean?
  - Kythe is a database of code annotations and references across a possibly multi-language codebase.
  - Each language must have its own “indexer” to analyze that language’s code.
  - It provides an index for data at one snapshot in time.
  - It’s used at Google to get cross-references for data across a very large code base.
  - We’ve developed a Lisp indexer plugin for Kythe so we can add Lisp data to a Kythe database.
- Why is it useful:
  - Indexing a code base, serving cross-references, creating call graphs, all with a static codebase.

Kythe's Schema: How we encode the graph

- The Kythe schema is robust enough to incorporate facets of many languages.
- Kythe creates Nodes to identify aspects of an object, VNames to uniquely identify those nodes, and edges between nodes.
- Take bordeaux-threads:threadp for example:
  - bordeaux-threads/impl-sbcl.lisp at master · sionescu/bordeaux-threads · GitHub
- We will look at the "object" variable on line 25:

```
25 (defun threadp (object)
26  (typep object 'sb-thread:thread))
```

Node

- The kind is the type of node we have, in this case we have a variable.
- The name is the name of the object in the code, as we would expect it's "object".
- The ticket is a URI encoding of the VName.
- The corpus is the root of the code repository your working in.

Example Kythe Output:

{ ticket: "kythe://corpus??lang=lisp?path=PATH#BORDEAUX-THREADS%3A%3AOBJECT%20%3AVARIABLE%20loc%3D%2825%3A16-25%3A22%29", kind: "variable", language: "lisp", name: "object", qualified_name: "object", location: { corpus: "corpus", path: "PATH/TO/bordeaux-threads/src/impl-sbcl.lisp", line_number: 25, line_number_end: 25, column_number: 16, column_number_end: 22 }, v_name: { signature: "BORDEAUX-THREADS::OBJECT :VARIABLE loc=(25:16-25:22)", corpus: "corpus", path: "PATH/TO/bordeaux-threads/src/impl-sbcl.lisp", language: "lisp" } }

Node

Given the form:

```
25 (defun threadp (object)
26  (typep object 'sb-thread:thread))
```

The node for *object* on line 25 is shown to the right.

The main sub-objects are location and VName.

1. Location, the file name and location within the file.
2. VName is a name that uniquely identifies this node.
    a. Each language has to make its own VName which makes intra-language edges difficult.

Example Kythe Output:

{ ticket: "kythe://corpus??lang=lisp?path=PATH#BORDEAUX-THREADS%3A%3AOBJECT%20%3AVARIABLE%20loc%3D%2825%3A16-25%3A22%29", kind: "variable", language: "lisp", name: "object", qualified_name: "object", location: { corpus: "corpus", path: "PATH/TO/bordeaux-threads/src/impl-sbcl.lisp", line_number: 25, line_number_end: 25, column_number: 16, column_number_end: 22 }, v_name: { signature: "BORDEAUX-THREADS::OBJECT :VARIABLE loc=(25:16-25:22)", corpus: "corpus", path: "PATH/TO/bordeaux-threads/src/impl-sbcl.lisp", language: "lisp" } }

Edges

Edges look like:

{source: node1, target: node2, edge_kind: edge_kind}

There is a second variable node for object on line 26 with edge_kind “ref” telling us that the node one line 26 references the node on line 25.

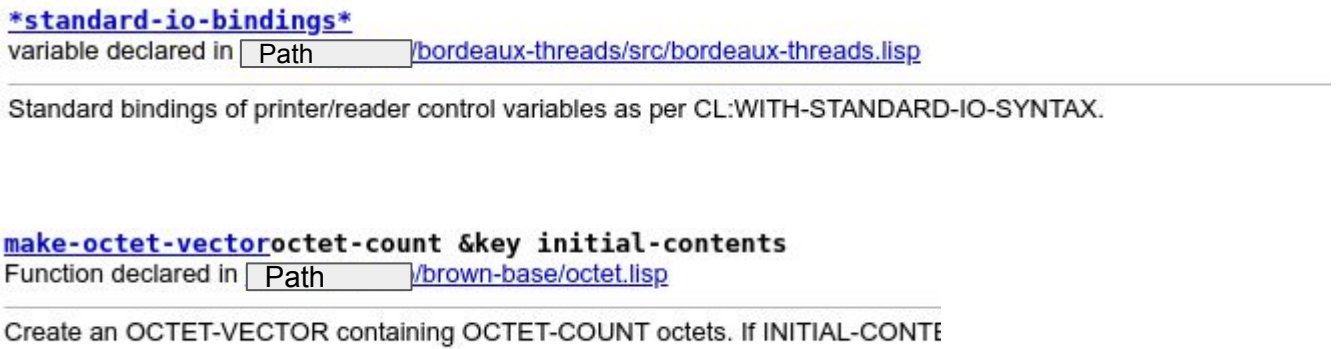
With proper IDE integration clicking on object on line 25 tells you there’s a cross reference on line 26 (as we see below):

```
25 (defun threadp (object)
26  (typep object 'sb-thread:thread))
```

For the full schema please reference: https://kythe.io/docs/schema/

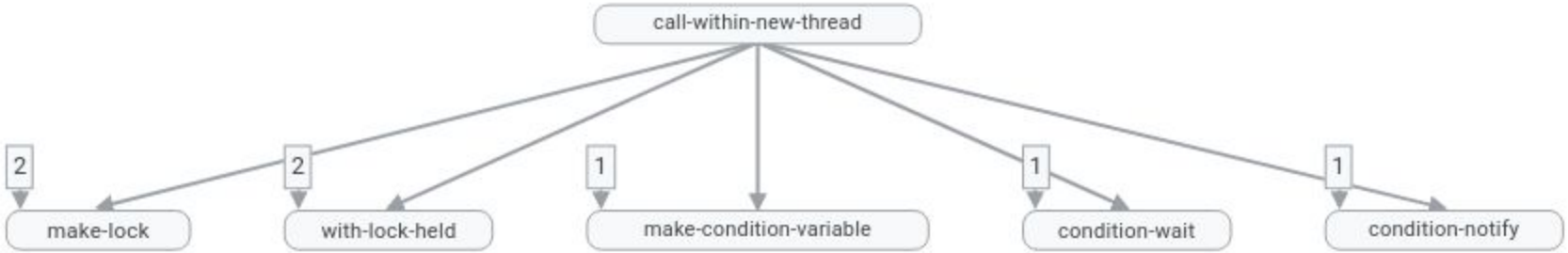
Web UI

Since docstring and lists of a functions variables are part of the schema we can display documentation:



More Web UI

We can also make call graphs



Taken from bordeaux-threads/src/impl-sbcl.lisp

Note the numbers are the numbers are the number of non-expanded places a function/macro is called in other files.

Running Kythe

Kythe is currently implemented to build and run with Bazel, the Google build system open sourced at: https://bazel.build/

There's a lisp plugin https://github.com/qitab/bazelisp

Kythe uses a dependency graph created by Bazel to know what files to compile.

It send the files to the language specific indexer in an analysis request.

How any language analyzes a file is up to the language itself.

In lisp, due to the nature of macros, we compile the file and use the cross-reference data, as well as the docstrings as we will discuss shortly.

Useful Tools

After running kythe the data can be sent into the Cayley Graph database:

> cayleygraph/cayley: An open-source graph database

for all of your querying and call-graph making wishes.

Kythe has its own command line tool:

https://kythe.io/docs/kythes-command-line-tool.html

Integration with LSP is simple, Kythe was designed with this partially in mind.

How do we Make the Nodes

- Call compile on a file with all of its dependencies.
- Built an AST of the file with source location info.
- Do a depth first search through the AST checking the who-calls database at each level.
  - This gives us non-inlined function references.
  - Macro and setf references for most things.
  - Docstrings
  - Global variable references.
- Since you compile the file, if the file is in the indexer binary you better hope no constants or structures have changed...

Basic Things We Miss

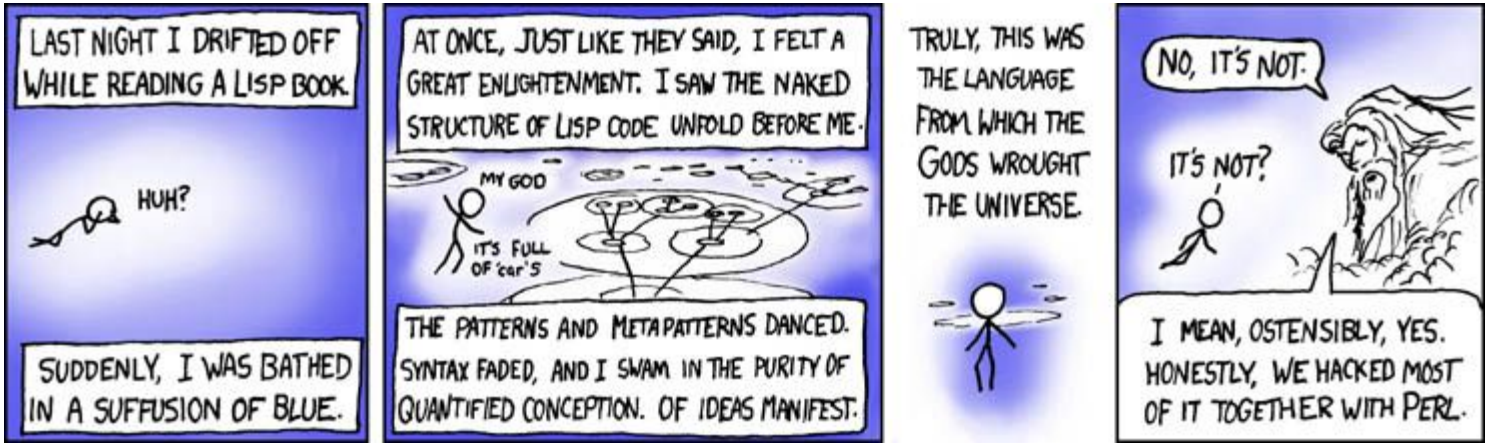
- Function argument bindings.
- Let, Flet and Labels bindings
- Loop binding
- These we can easily create parsers to figure out!
  - If you see (defun foo (bar baz) …) then ‘(bar baz) are the bound symbols
- Structure-object accessors
  - SBCL doesn’t use a traditional setf function for accessing structure fields
  - Thus the setf function is not in the who-calls database, so we can’t find it!
    - We iterate through the structure objects and add accessors to the who-calls database manually.

Lisp Difficulties: With great syntax...

- Lisp has no syntax, or it has all the syntax, you the dear listener make the syntax.
- In most languages, without Lisp macros, it's easier to tell what's being bound where.
- If I have the c++ function:

  ```
  int foo(int bar) {
      return ++bar;
  }
  ```

  I can say explicitly where bar is bound.
- Even with c++ macros I can say without to much trouble where each variable is bound.
- Lisp allows the user to define whatever syntax they want, whenever they want.



Basic Example

- How do I know what is bound in with-data-mutex?
- What's the difference between bindings and bodies?
- If &body is there our job's a little easier but it isn't always.
- What about anaphoric macros?

```
(defvar *process-data-mutex* (make-mutex))

(defmacro with-data-mutex ((mutex) &body body)
    `(let ((,mutex *process-data-mutex*))
        (sb-thread:get-mutex ,mutex)
         ,@body (sb-thread:release-mutex ,mutex)))

(defun process-data (data)
    (with-data-mutex (data-mutex)
        (format t "I have mutex ~a" data-mutex)
(print a)))
```

Inter-Language References

At Google we like to use protocol buffers

The message to the right defines a structure-object hello-world with an accessor proto2:hello-world-string.

We would like to know everywhere the lisp accessor proto2:hello-world-string is called.

If you know the VName of a node, you can make an edge from all of your accessors calls to the hello_world_string protobuf field.

The big issue is you have to know how to make your VNames.

hello_world.proto

```
syntax = "proto2";

package example;

message HelloWorld {

        optional string hello_world_string = 1;

}
```

Why Should I Use This?

- This lets you get cross-reference data for a many file, many language, repository.
  - Imagine having xrefs for all (open source) Lisp code world-wide.
- It can be used to produce call graphs.
- You have jump to definition outside an IDE.
- You can add the data to a graph store and query.

Future Work (my todo list)

- Open source the Lisp Kythe Indexer
  - This is on my roadmap, I promise.
  - While you're waiting try the C++ or Java indexers:
    - kythe/kythe: Kythe is a pluggable, (mostly) language-agnostic ecosystem for building tools that work with code.
- Have the Lisp Kythe Indexer index a CL other then SBCL.
  - SLIME uses the who-calls database for whatever common-lisp variant thats running, why can't we?
  - CCL will probably be nicer as its database is more robust.
- Make a generic macro parser
  - Code walker? Calling macrostep-expand a lot?
  - Ideas?

Acknowledgements

Jinwoo Lee and Andrzej Walczak for writing the majority of the indexer.

Eric Willison for writing much of the parser framework.

Ron Gut and Carl Gay for going over the conference paper, many cl’s,and much talking about indexing.

My wife Wenwen and daughter Lyra

You, for listening.

.

Citations:

Protocol buffers. https://developers.google.com/protocol-buffers. Accessed: 2020-02-10

Kythe: A pluggable, (mostly) language-agnostic ecosystem for building tools that work with code. https://kythe.io/, 2019. Accessed: 2020-02-10

Slime: The superior lisp interaction mode for emacs. Accessed: 2020-02-10.

Stelian Ionescu. Bordeaux threads. https://github.com/sionescu/bordeaux-threads

Robert Brown. Brown-base. https://github.com/brown/base