Scope Graphs

- Language
- Building the graph
  - AST
    - Qualifiers
  - Nodes
  - Edges
- Traversing the graph
  - Paths
  - Regex
  - Qualifiers
  - Match
- Sources

Scope graphs are an alternative approach for name resolution. A scope graph is an AST extended with:

- nodes representing scopes
- directed edges describing different relationships involving these scope nodes; including to other scope nodes and to original AST nodes. For example
  a. parent/child scopes
  b. declarations within a scope
  c. imports

This reduces name resolution to reachability, and has some advantages over the lookup table approach:

- Paths are recoverable
- Ambiguity
- Incrementality

Language

The goal was to build a prototype of a scope graph for a small language, which Jonathan wrote a parser for. This language is a subset of Flix and has the following constructs:

- names (qualified and unqualified)
- accessibility modifiers
- expressions
  - literals, names, function application, sequencing, let binds
- types
  - base types, type application/vars
- defs
- modules
- uses/imports with aliases/renamings
- type aliases
- enums

Building the graph

AST

The AST is designed to distinguish **declarations** and **references**. Here, declarations refer to the entire AST node, so we also introduce **handles**.

- **Declaration -** refers to an entire AST node that introduces a new name into the current scope (module, enum, type alias, etc)
- **Handle -** the name introduced by a declaration, located within the declaration
- **Reference -** the name of a declaration, used to refer to the declaration from elsewhere within the AST

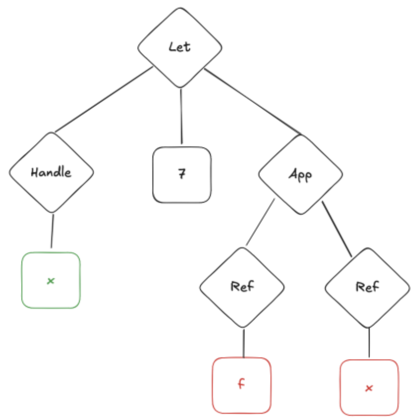
Thinking in terms of the IDE:

- If you ctrl click a reference, you should navigate to the matching handle.
- If you ctrl click a handle, you should get a drop down of references to it.

Every identifier in the AST is either a handle or a reference. And every declaration has a handle.

Example:

```
let x = 7;
f(x)
```

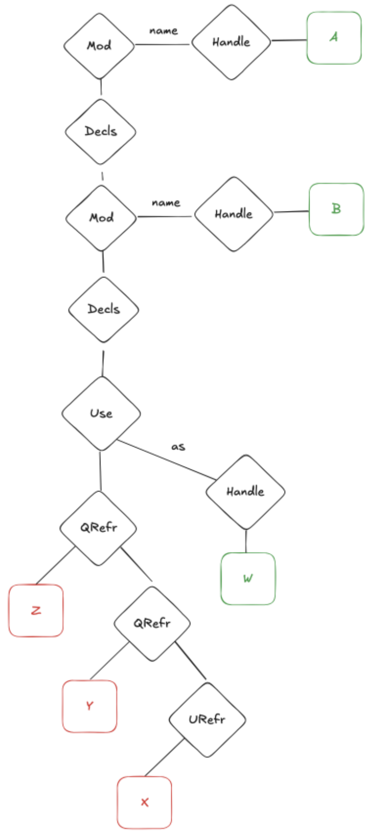


Qualifiers

- References are inductive types. If a reference has qualifiers, then every qualifier is itself a reference.
- Handles are just identifiers. If a module has qualifiers, then those qualifiers are desugared to multiple module declarations.

Example:

```
mod A.B {
  use X.Y.Z as W
}
```



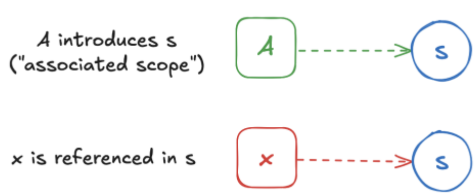
`W` is a handle 🙂

type variables complicate things

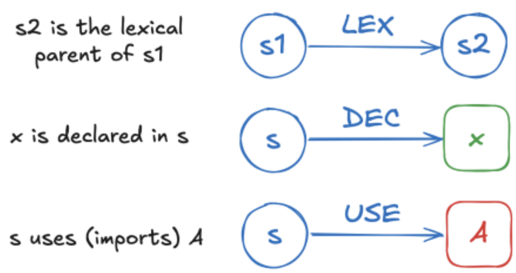
Next we we add scope nodes and directed edges, which allow us to construct paths from references to handles/declarations.

Nodes

- **Declarations** introduce scopes
    - “associated scopes”
    - global vs local
        - global: associated scopes of `enum` s or `module` s, which can be accessed globally via a `use` statement.
        - local: associated scopes of other declarations, such as `def` or `let` , which can never be accessed outside of the declaration body.
- **References** are declared in scopes



Edges



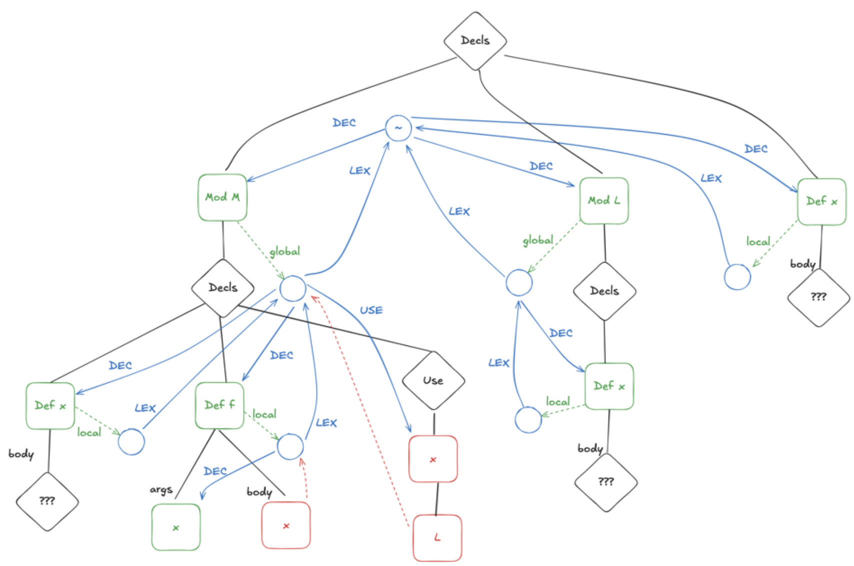
Example:

```
def x(): Int = ???

mod L {
  def x(): Int = ???
}

mod M {
  use L.x

  def x(): Int = ???
  def f(x: Int): Int = x
}
```



Traversing the graph

Paths

Once the graph has been built, we can resolve names by finding paths over the scope edges.

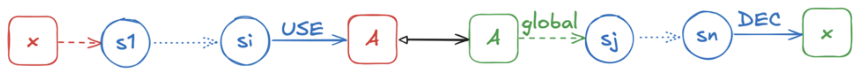
A path looks something like:

Note that the blue `DEC` / `LEX` / `USE` edges are the only "scope edges" that count in a path. The dotted arrows indicate relationships, but they are not officially part of the "path," which is important once we look at regular expressions.

If such a path exists, then we can say "x refers to x" using this notation:

The path from s1 to sn might:

- be empty.
- be just some number of LEX edges.
- include a USE edge to something with a global scope (module or enum), in which case we recursively find a path to resolve the symbol in the `use` statement, and then continue the search:



If we have a USE edge to a declaration with a local scope (def or enumCase), this can be the target, but we cannot continue the search:

Regex

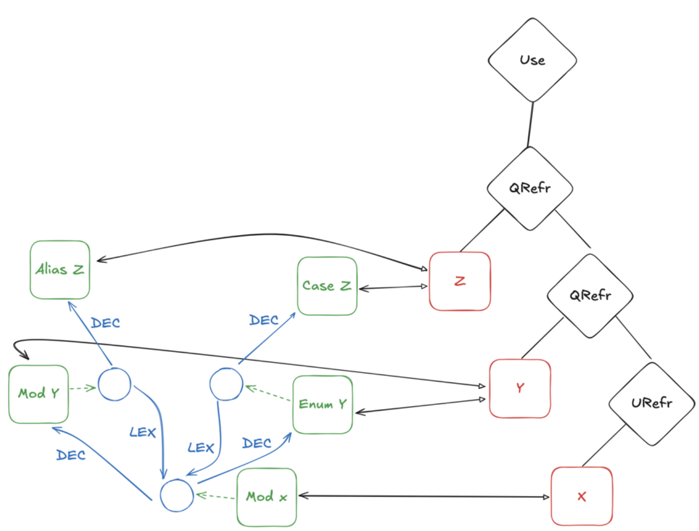
There can be many declarations that “match” a reference, each with their own path. In order to formalize the scoping rules of a language, you need two things:

1. A regular expression (over the alphabet `{USE, DEC, LEX}` that describes valid paths.
    a. ex: `LEX* USE* DEC?` allows transitive imports, but not lexical parents of imported modules.
2. An ordering over `{USE, DEC, LEX}` to decide between multiple valid paths.
    a. We are looking for the *shortest* matching path.
    b. Note: two equal length paths may indicate a programmer error (ambiguous reference) or potentially something valid (importing an enum vs a mod).

Qualifiers

When we encounter qualified references, we resolve the qualifier recursively, similar to a `use`, then continue searching.

```
mod X {
  enum Y {
    case Z
  }
  mod Y {
    type alias Z
  }
}
```



Match

When we reach a declaration, how do we know if it matches?

1. The names have to be the same.
2. The "kind" (for lack of a better word) has to match.
   a. If the reference appears in a type signature, then an enum or a type alias declaration could be a match.
   b. If the reference appears in an expression, then a def or enum case declaration could be a match.
   c. I have this half implemented in an ad hoc way. I don't think I've thought it through well enough.
   d. If the reference appears in a qualifier or an import, it could be multiple things. (`Kind.Any` ?)
3. The visibility modifier has to be okay.
   a. `private` declarations should not be accessible if we've crossed a `USE` edge.
   b. I did not get to this.

Something I considered but didn't explore would be to have filters of type `Decl → Bool` that are accumulated during search. So, if we cross an import, we `and` in a filter that disallows `private` declarations. This could be an alternative way to represent "specific" imports.

Sources

These are the things I referenced while working on this:

- https://drops.dagstuhl.de/storage/00lipics/lipics-vol313-ecoop2024/LIPIcs.ECOOP.2024.47/LIPIcs.ECOOP.2024.47.pdf
- https://drops.dagstuhl.de/storage/01oasics/oasics-vol109-evcs2023/OASIcs.EVCS.2023.32/OASIcs.EVCS.2023.32.pdf
- https://eelcovisser.org/talks/2017/2017-06-curryon/scope-graphs-curryon-2017-06-20.pdf