How to compile:

1. Follow the prereqs in the readme
2. ./cabal-hls-install hls
3. Take note of where cabal builds the binary, by default on Linux this is $HOME/.cabal/bin

How to use your hls binary:

1. If you’re using VSCode, install the Haskell extension
2. Navigate to the extension settings, and find the line that says Haskell: Server Executable Path
3. Add the file path to your binaries in the associated box.
4. If it’s in the cabal default location, it should look like this:

${HOME}/.cabal/bin/haskell-language-server

1. Now restart VSCode, and make sure your standard plugins, such as details on hover and comment evaluation, work as expected.

How to enable the example plugins:

1. The example plugins are located in plugins/default/src/Ide/Plugin, as Example.hs and Example2.hs
2. To enable them easily, go to exe/Main.hs
3. Change line 21 from:

defaultMain args (idePlugins withExamples)

    to:

        defaultMain args (idePlugins True)

1. Now recompile and look at all the cool example features
2. To turn off the examples, simply revert the changed line of code and recompile

How to add your plugin:

1. Your module should be named something like Ide.Plugin.TestPlugin, and be located in plugins/default/src/Ide/Plugin/TestPlugin.hs
2. In exe/Plugins.hs, add your plugin as an import
3. Later in that file, add your plugin to the list of other plugins
4. Go to haskell-language-server.cabal
5. Add your plugin to the big list of plugins that all start with flag/common. Your block should look something like:

        common test-plugin

            hs-source-dirs: plugins/default/src

            other-modules: Ide.Plugin.TestPlugin

1. In the same file, add your plugin to the import list that comes after all the plugin blocks.
2. Now you can compile hls and test out your features in VSCode

NOTES:

[This page](http://www.aosabook.org/en/ghc.html) gives a great rundown of how GHC works, and its structure. It is all interesting, but of specific relevance to this project are the sections on parsing, renaming, and typechecking, and later the “No Symbol Table” section.

Relevant Packages:

* [SrcLoc](https://hackage.haskell.org/package/ghc-8.10.2/docs/SrcLoc.html)
  + Types for positions in source files. Includes Location type
* [HsSyn](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsSyn.html)
  + Abstract syntax tree types. Some of the structure is detailed in a tree below. The following packages with the Hs prefix contain the info for that tree
* [HsDecls](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsDecls.html)
  + Types for various declarations, including type/class, instance, deriving, signatures, etc.
* [HsBinds](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsBinds.html)
  + Types for local bindings. Includes function bindings and signatures
* [HsExtension](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsExtension.html)
  + Contains info about the 3 compiler passes that use HsSyn
* [HsTypes](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsTypes.html)
  + Haskell types. Contains info about function types, signatures, wildcard binders
* [TcRnTypes](https://hackage.haskell.org/package/ghc-8.2.2/docs/TcRnTypes.html)
  + Types for typechecked/renamed environment
* [Development.Ide.Core.RuleTypes](https://hackage.haskell.org/package/hie-core-0.0.1/docs/Development-IDE-Core-RuleTypes.html)
  + Some compiler types. Allows access to the result of the typechecker
  + E.g., the line “use TypeCheck nfp” will produce a Maybe TcModuleResult
  + A TcModuleResult contains, among other things, a ParsedModule, a RenamedSource, and a TcGblEnv
* [Development.Ide.GHC.Compat](https://hackage.haskell.org/package/ghcide-0.6.0/docs/Development-IDE-GHC-Compat.html)
  + Access to A LOT of compiler info, and the ability to run compiler steps
  + Contains types for parsed/renamed/typechecked modules
* [RnNames](https://hackage.haskell.org/package/ghc-8.10.2/docs/RnNames.html)
  + Functions that produce an RnM (renaming monad). Includes functions to get import usage
* [HsImpExp](https://hackage.haskell.org/package/ghc-lib-parser-8.8.2/docs/HsImpExp.html)
  + Types for import/export declarations
* [HscTypes](https://hackage.haskell.org/package/ghc-8.10.2/docs/HscTypes.html)
  + More compiler types, such as the environment. Has module and package info
* [Type](https://hackage.haskell.org/package/ghc-8.10.2/docs/Type.html#t:Type)
  + The Type type :p
  + Also Kind type (Just a synonym for Type)
  + Lifted type:
    - Contains “bottom” aka undefined.
    - Has kind \*
  + Unlifted type:
    - Doesn’t contain bottom, essentially cannot be non-terminating or failing
    - Has kind #
  + Boxed type:
    - A value represented by a pointer to a heap object
    - Lifted types are boxed, but not necessarily vice versa
  + [Unboxed type](https://wiki.haskell.org/Unboxed_type):
    - Raw values
    - Has kind #
    - Represented by appending a # suffix, e.g. Int#
    - To use these, use language extension MagicHash, and import GHC.Prim
  + [Algebraic type](https://wiki.haskell.org/Algebraic_data_type):
    - A type with constructors, declared with data or newtype
  + Data:
    - Declared with data or a boxed tuple
  + Primitive:
    - Built in types
  + PredType:
    - A type that is a predicate, aka the thing that goes before => in a type
  + [Levity polymorphism](https://stackoverflow.com/questions/35318562/what-is-levity-polymorphism)
    - The ability for functions to be polymorphic over lifted and unlifted types
    - I.e., functions accept types with kinds \* and #, instead of just \*
  + More reading on some GHC type and kind stuff [here](https://gitlab.haskell.org/ghc/ghc/-/wikis/intermediate-types)
    - Has a kind tree
    - Kinds = Types
    - Shows how some kinds are represented in the code
    - Coercion kinds, aka things that denote types as able to be cast to each other
  + Ability to split function args and result types
  + Type equality checking
* [Var](https://hackage.haskell.org/package/ghc-8.10.2/docs/Var.html)
  + Typed Names, may contain type variables which have a kind
  + Bunch of synonyms to convey extra info
  + Can be global or local
* [Id](https://hackage.haskell.org/package/ghc-8.10.2/docs/Id.html#v:Id)
  + Typed Names, with a bit of extra info
* [Name](https://hackage.haskell.org/package/ghc-8.10.2/docs/Id.html#v:Id)
  + names with scoping and binding resolved. Contain some info about where they’re from
  + Also contains class NamedThing, which allows access to the Name and Location of various datatypes
* [FV](https://hackage.haskell.org/package/ghc-8.10.2/docs/FV.html)
  + Free variables
* [VarSet](https://hackage.haskell.org/package/ghc-8.10.2/docs/VarSet.html)
  + Sets of variables, non-deterministic
  + Contains expected functions for sets
  + DVarSet for a deterministic one
* [TyCon](https://hackage.haskell.org/package/ghc-8.10.2/docs/TyCon.html)
  + Type constructors, e.g. data, type, newtype, class

HsSyn:

Data HsModule name =

hsmodImports :: [[LImportDecl](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsImpExp.html#t:LImportDecl) name]        --Constructor used for the explicit imports plugin

hsmodDecls :: [[LHsDecl](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsDecls.html#t:LHsDecl) name]            --Constructor I want for type signatures (I think)

  -> type  LHsDecl id = Located (HsDecl id)

        -> data HsDecl id = SigD (Sig id)        --Constructor for signature declarations

      -> data TypeSig [Located (IdP pass)] (LHsSigWcType pass)

--Constructor for normal type signatures

         ->type family IdP p           --Maps the "normal" id type for a given pass

                            --Parsed, renamed, or typechecked

            ->type LHsSigWcType pass =     HsWildCardBndrs pass (LHsSigType pass)

--Located Haskell Signature Wildcard Type

          ->type LHsSigType pass = HsImplicitBndrs pass (LHsType pass)

            --Located Haskell Signature Type

       ->type LHsType pass = Located (HsType pass)

                --Located Haskell Type

    -> data HsType pass =    --There’s a lot here. [Link](https://hackage.haskell.org/package/ghc-8.4.3/docs/HsTypes.html#t:HsType)

Location stuff is [here](https://hackage.haskell.org/package/ghc-8.10.2/docs/SrcLoc.html#t:SrcSpanLess). Some useful things are:

* getLoc :: HasSrcSpan a => a -> SrcSpan
* unLoc :: HasSrcSpan a => a -> SrcSpanLess a

Deconstructing the explicit imports plugin:

There are a few terms we need to know to understand plugins.

First, a codeLens (from [here](https://vscode.rocks/codelens/)):

“As described by Wade Anderson, CodeLens are “actionable contextual information interspersed in your source code”. They’re links that you see above lines in your vs code editor that give actions as well as extra information to your codebase. They are similar to the decorator API to give more context to your code, but you can click on CodeLens to take actions based on it.”

A few other things, like commands and rules, are defined in this [tutorial](https://github.com/pepeiborra/hls-tutorial). Be aware, much of the code here is outdated, but the descriptions give a good sense of plugin anatomy.

Tl;dr: Code lenses are the little lines of text that appear above code that you can click on to do something. E.g. making a single import statement explicit. Code actions are things that appear in those little yellow lightbulbs, and can act as fixes for warnings and errors. E.g. Making every import explicit. Rules add targets to the Shake build graph. This means that they are included in the ideState that gets passed around the plugin, and the RuleResults can be accessed by the ‘use’ combinators.

Helper functions (These are at the bottom of the module):

* runIde :: IdeState -> Action a -> IO a

runIde state = runAction "importLens" state

* Runs ide actions
* within :: Range -> SrcSpan -> Bool

within (Range start end) span =

  isInsideSrcSpan start span || isInsideSrcSpan end span

* Is a range within a span

lensProvider:

* For each implicit import, return a code lens containing the explicit import
* Just nfp <- uriToNormalizedFilePath $ toNormalizedUri \_uri
  + Normalizes a file path
* mbMinImports <- runAction "" state $ useWithStale MinimalImports nfp
  + mbMinImports :: Maybe (MinimalImportsResult, PositionMapping)
  + Gets the minimal imports and their positions by extracting the result of our rule.
* commands <-

              sequence

                [ generateLens pId \_uri edit

                  | (imp, Just minImport) <- minImports,

                    Just edit <- [mkExplicitEdit posMapping imp minImport]

                ]

* This block creates the lenses, using the edits created by mkExplicitEdit
* return $ Right (List $ catMaybes commands)
  + The final result
* Note that the fail states do this
  + return $ Right (List [])

codeActionProvider:

* If there are any implicit imports, provide one code action to turn them all into explicit imports.
* pm <- runIde ideState $ use GetParsedModule nfp
  + Parses the module
* insideImport is a bool
  + False if pm isn’t this
    - Just ParsedModule {pm\_parsed\_source}
  + Otherwise determined by
    - locImports <- hsmodImports (unLoc pm\_parsed\_source),

rangesImports <- map getLoc locImports ->

any (within range) rangesImports

* Gets the imports, gets the location ranges from those imports, checks if any of them are within the range for the action
* minImports <- runAction "MinimalImports" ideState $ use MinimalImports nfp
  + Gets the minImports by running the action
* edits =

                [ e

                  | (imp, Just explicit) <-

                      maybe [] getMinimalImportsResult minImports,

                    Just e <- [mkExplicitEdit zeroMapping imp explicit]

                ]

* Creates the edits using the results of minImports
* aExplicitImports = InR CodeAction {..}

              \_title = "Make all imports explicit"z

              \_kind = Just CodeActionQuickFix

              \_command = Nothing

              \_edit = Just WorkspaceEdit {\_changes, \_documentChanges}

              \_changes = Just $ HashMap.singleton \_uri $ List edits

              \_documentChanges = Nothing

              \_diagnostics = Nothing

              \_isPreferred = Nothing

              \_disabled = Nothing

* All of the parameters for a CodeAction. \_edit and \_changes are the significant ones here
* return $ Right $ List [caExplicitImports | not (null edits)]
  + The final result
* return $ Right $ List []
  + Again the branches are this

minimalImportsRule

* The rule that figures out the minimal imports and adds them to the build graph
* tmr <- runAction "importLens" state $ use TypeCheck nfp
  + Get the typechecking artifacts from the module
* hsc <- runAction "importLens" state $ use GhcSessionDeps nfp
  + We also need a GHC session with all the dependencies
* (imports, mbMinImports) <- extractMinimalImports hsc tmr
  + Use the GHC api to extract the "minimal" imports
* importsMap = Map.fromList

          [ (srcSpanStart l, T.pack (prettyPrint i))

            | L l i <- fromMaybe [] mbMinImports

          ]

* Creates a map where the key is the start of the location range, and the value is the import declarations converted into Text
* Info on [Map](https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html), [Text](https://hackage.haskell.org/package/text-1.2.4.1/docs/Data-Text.html) (T)
  + Tldr; prettyPrint makes i a string, T.pack makes a string a text
* l :: SrcSpan
* i :: ImportDecl GhcRn
* res = [ (i, Map.lookup (srcSpanStart (getLoc i)) importsMap)

          | i <- imports

        ]

* This gets the imports, and finds them in the map of minimal imports from the previous step (which uses locations as keys). We are left with a list of pairs of imports and Maybe Text values.
* return ([], MinimalImportsResult res <$ mbMinImports)
  + The final result
  + (<$) :: a -> f b -> f a
    - Replace all locations in the input with the same value. The default definition is fmap . const
  + This creates an Action, with an empty list of FileDiagnostics, and a Maybe MinimalImportsResult

extractMinimalImports

* Use the ghc api to extract a minimal, explicit set of imports for this module
* extractMinimalImports ::

  Maybe HscEnvEq ->

  Maybe TcModuleResult ->

  IO ([LImportDecl GhcRn], Maybe [LImportDecl GhcRn])

* If either of those maybes are Nothing we get this:
  + return ([], Nothing)
* let tcEnv = tmrTypechecked

      (\_, imports, \_, \_) = tmrRenamed

      ParsedModule {pm\_parsed\_source = L loc \_} = tmrParsed

      span = fromMaybe (error "expected real") $ realSpan loc

* Extract the typechecked environment, the imports, the parsed module, and the span
* gblElts <- readIORef (tcg\_used\_gres tcEnv)
  + Some GHC nonsense ¯\\_(ツ)\_/¯
* let usage = findImportUsage imports gblElts
  + Uses a nifty GHC function to find the usage of our imports in the environment
* (\_, minimalImports) <- initTcWithGbl (hscEnv hsc) tcEnv span $ getMinimalImports usage
  + Gets the minimal imports from usage, then uses some more shenaniganry to ??initialize a typecheck??
* return (imports, minimalImports)
  + The final result. A pair of the original imports and the new minimal ones

mkExplicitEdit

* Creates the text edit for the explicit import
* | ImportDecl {ideclHiding = Just (False, \_)} <- imp = Nothing
  + The case where an import is already explicit
* | not (isQualifiedImport imp),
  + Make sure the import is not qualified
* RealSrcSpan l <- src,
  + Get the location of the import
* L \_ mn <- ideclName imp,
  + Get the module name of the import
* mn /= moduleName pRELUDE,
  + Make sure that the module is not Prelude, we don’t want explicit imports for that
* Just rng <- toCurrentRange posMapping $ realSrcSpanToRange l
  + Get the range for the edit
* Just $ TextEdit rng explicit
  + Create the text edit using the range and the provided text
* | otherwise =

    Nothing

generateLens

* let title = \_newText importEdit
  + Simply turns the TextEdit into a Text to act as a title
* \_xdata = Nothing
  + No extra data
* edit = WorkspaceEdit (Just editsMap) Nothing
  + Creates a WorkspaceEdit which replaces the import declaration with an explicit one
* editsMap = HashMap.fromList [(uri, List [importEdit])]
  + Creates a HashMap where the file’s uri is the key, and the value is a List containing the importEdit
* \_arguments = Just [toJSON $ ImportCommandParams edit]
  + Gets the command parameters from the edit and turns them into JSON to use as arguments
* \_command = Just $ mkLspCommand pId importCommandId title \_arguments
  + Creates a command using the plugin ID, the command ID, the title, and the arguments
* return $ Just CodeLens {..}
  + The final result
  + CodeLens is {\_range, \_command, \_xdata}

Putting it all together:

importCommandId :: CommandId

importCommandId = "ImportLensCommand"

* This is simply a name for our command

Descriptor

* (defaultPluginDescriptor plId)
  + Uses the default descriptor, passing it our plugin ID
* pluginCommands = [importLensCommand]
  + The sole command for our plugin
* pluginRules = minimalImportsRule
  + We give it the rule we made
* pluginHandlers = mconcat

        [

          mkPluginHandler STextDocumentCodeLens lensProvider

        , mkPluginHandler STextDocumentCodeAction codeActionProvider

        ]

* We have two handlers, one for our lens and one for our action

importLensCommand

* PluginCommand importCommandId "Explicit import command" runImportCommand
  + Create a command with an ID, a descriptor, and the actual handler

runImportCommand

* \_ <- sendRequest SWorkspaceApplyEdit (ApplyWorkspaceEditParams Nothing edit)

(\\_ -> pure ())

* Triggers a workplace edit using the edit that gets passed in, which should be the one we made earlier
* return (Right Null)
  + We just wanted to do the edit, so the result can be nothing

Overall, our functions form a hierarchy something like this:

Descriptor

* importLensCommand
  + importCommandId
  + runImportCommand
* minimalImportsRule
  + extractMinimalImports
* lensProvider
  + generateLens
  + mkExplicitEdit
* codeActionProvider
  + mkExplicitEdit

So, we have a lens provider and an associated command. These work together to create the lens which allows us to make an import explicit. We also have an action provider which gives us the lightbulb action that makes every import explicit. These both use the same edit, from the mkExplicitEdit function. We also provide a rule, which is what actually figures out the minimal imports, and adds them to the ide state. Our providers use ‘use’ to access the results of this rule.