Setup.hs

Exercises

Lecture 1 - GHCup

Installation

Your first goal for these exercises is to install GHCup, which works a bit differently depending on your operating system. As long as you can run the command `ghcup --version`, you can move on to the next part of these exercises (**Listing and Changing Versions**).

You can find the full instructions online here: https://www.haskell.org/ghcup/install/

These instructions include any system requirements, including which distro packages are required on Linux.

Linux (Including Windows Subsystem for Linux)

Before installing GHCup, you must ensure you have installed the following distro packages (using `apt` or similar). I'm printing the list given for Ubuntu. It might change a bit depending on your distribution:

```
build-essential
curl
libffi-dev
libffi8ubuntu1
libgmp-dev
libgmp10
libncurses-dev
libncurses5
libtinfo5
```

The `curl` command below will also pause and print the list of requirements before asking you to proceed with installation. So if required packages are different for your distro, you'll be able to see.

Once you have those dependencies met, installation is easy. You just need to run this `curl` command.

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

There will be a series of prompts. You should make sure GHCup is added to your path and that Haskell Language Server is installed. When it comes to allowing GHCup to manage Stack's GHC versions, I indicated "no". We'll see the consequences of this later. You're welcome to choose the default option of "yes".

This process should modify your `.bashrc` file to update your `PATH` to include the locations of the installed programs. So after the installation finishes, you should be able to run the command `ghcup --version` after re-sourcing your `.bashrc` file.

MacOS

On Mac, you basically only need to run the `curl` command. Dependencies should either already be installed, or should be added as part of the process of running this script.

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

Windows

Doing a pure Windows build is a bit different. You'll be working in Windows Powershell, which has more requirements, so the command is more complicated. Here it is, spaced over multiple lines so you can see what's going on (we'll have a copyable one-line version further down, and the GHCup website also has a copyable version).

```
>> Set-ExecutionPolicy Bypass -Scope Process -Force; `
  [System.Net.ServicePointManager]::SecurityProtocol = `
  [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; `
  try { Invoke-Command -ScriptBlock `
    ([ScriptBlock]::Create((Invoke-WebRequest `
    https://www.haskell.org/ghcup/sh/bootstrap-haskell.ps1 `
    -UseBasicParsing))) -ArgumentList $true } `
  catch { Write-Error $_ }
```

Running this command will bring up a couple different prompts. You can just press "enter" and accept the default options. For me, it took several minutes for all the scripts to run and a second window opened at one point. When everything was finished, the second window gave instructions to close both terminals and *then reopen a new Powershell terminal.* After opening a new terminal, I was able to run `ghcup --version` successfully.

Here's the full command on "one line" if you want to copy it all at once instead of breaking it up into different lines (as mentioned, this command is also available through the GHCup website: https://www.haskell.org/ghcup/):

```
Set-ExecutionPolicy Bypass -Scope Process
-Force;[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; try { Invoke-Command
-ScriptBlock ([ScriptBlock]::Create((Invoke-WebRequest
https://www.haskell.org/ghcup/sh/bootstrap-haskell.ps1 -UseBasicParsing))) -ArgumentList $true
} catch { Write-Error $_ }
```

Listing and Changing Versions

When you install GHCup, it will, by default, install all the recommended versions of the different tools mentioned in the lecture (GHC, Cabal, Stack, HLS). Use the `ghcup list` command to see all the available versions. You should see that one version of each program is both installed and selected (indicated by double check marks or the green letters **IS** on Windows).

For the following programs, run the command to check its version and verify that you see the same version that was indicated as "selected" by the `list` command.

```
>> ghc --version
>> cabal --version
>> stack --version
```

Now try installing and setting a different version of GHC. You can do this with two commands. I'll use version `9.4.3` as an example (at the time of writing, GHC `9.2.5` is the recommended version).

```
ghcup install ghc 9.4.3
ghcup set ghc 9.4.3
```

Verify that this worked by checking the version again:

```
ghc --version
The Glorious Glasgow Haskell Compilation System, version 9.4.3
```

Then you can set the version back once again to the recommended version.

If you're on Linux or Mac, you can also try going through this process with the textual user interface (TUI):

```
ghcup tui
```

You should then navigate up to the GHC version you want and press `i` to install it, and then `s` to set it. You can press `q` to close the interface and check the version once again.

Now you're ready to move on to the next lecture!

Lecture 2 - GHC

To start, make sure you've downloaded and unzipped the `Setup-hs.zip` file from the lecture! The link for it should be right under where you got this exercises document. We'll be working in this project directory from most of the lectures from now on. For this lecture, all your code will be in the `Lecture2` directory of the project.

Using runghc

Now, take a look at `Lecture2/HelloWorld.hs`. It has the basic structure we talked about in the lecture. Use `runghc` on this file and your console should simply print out the `"Hello World!"` message.

```
>> cd Lecture2
>> runghc HelloWorld.hs
Hello World!
```

Building with GHC

Now try compiling the file with `ghc`.

```
>> ghc HelloWorld.hs
```

You should observe that this creates 3 new files in the `Lecture2` directory: `Main.hi`, `Main.o`, and `HelloWorld` (this will be `HelloWorld.exe` on Windows). The last of these is an executable that you can run! So run it and observe that it prints the same message!

```
>> ghc HelloWorld.hs
>> ./HelloWorld
Hello World!
```

Now let's try that again, except let's ensure that all our generated files are placed in a nice location (the `artifacts` directory) instead of cluttering up our working directory. So delete all the generated files and then compile like so. Observe that we use a different name for the executable file:

```
>> rm Main.hi Main.o HelloWorld
>> mkdir -p artifacts
>> ghc HelloWorld.hs -hidir ./artifacts -odir ./artifacts \
  -o ./artifacts/Hello
>> cd artifacts
>> ./Hello
Hello World!
```

As another note on Windows, you should use a backtick `` ` `` instead of backslash `\` for continuing your command to another line.

Using Other Modules

Now go into `HelloWorld.hs`. Instead of using the string `"Hello World!"`, import the `Library` module and use its `greeting` string. You should be able to recompile the program and see that the output has changed:

```
>> ghc HelloWorld.hs -hidir ./artifacts -odir ./artifacts \
  -o ./artifacts/Hello
>> cd artifacts
>> ./Hello
Hello Library!
```

As a quick experiment, try `runghc Library.hs`. It **fails**, since this module has no `main` function!

Now try importing `MyStrings` instead of `Library` to import the `greeting`. This second module actually lives in the `Lecture2/Lib` directory. Try compiling it and you should see that it fails!

When importing a module from a different directory, `ghc` expects you to use the directory names as prefixes. So you need to make two changes:

1. In `HelloWorld.hs`, use `import Lib.MyStrings` instead of `import MyStrings`
2. Go into `Lib/MyStrings.hs` and change the module header (at the top of the file) to read `Lib.MyStrings` instead of `MyStrings`.

Now you should finally be able to run the code!

```
>> ghc HelloWorld.hs -hidir ./artifacts -odir ./artifacts \
  -o ./artifacts/Hello
>> cd artifacts
>> ./Hello
Hello MyStrings!
```

One final thing to try...go into the `Executable` directory and try running the `HelloWorld.hs` file in there.

```
>> cd Lecture2/Executable
>> runghc HelloWorld.hs
```

It will **fail**, because it relies on importing the `Library` module, and it can't search the directory above where `Library` is located! We can't import, for example, `../Library` in our Haskell code.

Now there are different options you can use with `ghc` to import modules in this kind of irregular structure. But for most users, especially beginners, it's not worth going into too much depth with the details of GHC because we have a couple other tools that deal with all these issues for us! In the next lecture, we'll learn about these tools!

Lecture 3 - Stack and Cabal

Building the Code

We'll be working from the `Lecture3` directory from the project root. Within this directory, there are two different Stack projects, `project1`, and `project2`. To start, head into the `project1` directory, and start building the code.

```
cd Lecture3/project1
stack build
```

As you can see from looking at the `stack.yaml` file in this directory, `project1` uses the Stack resolver 19.20, which corresponds to GHC `9.0.2`, which you have (probably) not installed yet. Because of this, the build project will take some extra time to download and install this new version of GHC. But it should eventually finish.

Now open up `project1/src/Lib.hs`. Modify the `printedString` expression so that it uses the number `"5"` instead of the given string. Then try running `stack build` again. You should see that the compiler throws an error.

Running an Executable

Now go into the `project2` folder. This uses a resolver for GHC `9.2.5`, which was the recommended version at the time of writing. So in theory, you could build the project without having to download a new version of GHC. But it depends on the option you selected when installing GHCup! If you selected "no" as I did on allowing GHCup to manage Stack's GHC versions, it will still re-download the compiler.

```
cd Lecture3/project2
stack build
```

Now run the project's executable with the following command:

```
stack exec project2-main
```

This should print the string `"Lecture 3 - Project 2"`. You can see this string in the file `app/Main.hs`, which has a `main` function. Modify the string in this file (perhaps `"Lecture 3 - Modified!"`). Then run the executable again. You should actually see the same result!

```
stack exec project2-main
Lecture 3 - Project 2
```

To update the executable with your changes, you need to build your code again.

```
stack build
stack exec project2-main
Lecture 3 - Modified!
```

Making a New Project

For the last part of these exercises, try making your own project in this directory:

```
cd Lecture3
stack new project3
```

After it's done creating the project, build and run it! It should print the string `"someFunc"` (unless the default Stack template has changed lately).

```
cd project3
stack build
stack exec project3-exe
someFunc
```

To modify the printed string, you'll need to go into the `src/Lib.hs` file and modify the `someFunc` expression.

As suggested in the lecture, I do recommend you delete the `package.yaml` file unless you're sure you want to use Hpack (https://hackage.haskell.org/package/hpack). Otherwise you might make modifications to the `.cabal` file, only to have them overwritten by the regeneration behavior!

Now you're ready to move on to the next lecture!

Lecture 4 - Executables

For the next three lectures, we'll be working with the `Quiz` project, which has its own directory (`Setup-hs/Quiz`).

Using a Dependency

Open up the `Quiz.cabal` file and look for the `run-quiz` executable. You should see that its source directory is `app`, and the main file is `RunQuiz.hs`. If you open the file, you'll see that it just prints a message. We want to change this so that it instead uses the `runQuiz` function from the `src/Runner.hs` file. Notice that its type is `IO ()`, just like `main`, so you can do a direct substitution. Of course, you'll need to import the `Runner` module.

Try building the code:

```
stack build
```

You'll find it doesn't work, because it's missing a dependency. Add the library (`Quiz`) to the executable's dependency list. Then you'll be able to build and run the executable!

```
stack build && stack exec run-quiz
```

Creating Your Own Executable

Now make a new executable, called `print-sum`. You can copy most of the definition (except the executable name and file name) from `run-quiz`. Its main file should be `app/PrintSum.hs`. Then you should also include the `split` package as a dependency.

Inside the file, fill in the `main` function like so:

```
module Main (main) where

import Data.List.Split (splitOn)

main :: IO ()
main = do
  putStrLn "Enter an equation {x} + {y}"
  input <- readLine
  let [a, b] = splitOn " + " input
  print (read a + read b)
```

You should now be able to build and run the executable:

```
stack build && stack exec print-sum
```

Lecture 5 - The Library

Adding a Module to the Library

For this lesson, let's take a closer look at the `Runner` module. Right now we're using `readMaybe` to parse the user's input as an integer (see the part of the `runQuestion` where we call `parseInt`). But we might want to pursue alternative ways to parse the input.

For example, if you use `run-quiz` and enter a very large or negative number for the answer to the questions, you'll simply get marked as incorrect even though these can't be the answers. We would like to restrict the user inputs to just the integers `1-4`. If they enter anything that doesn't look like an integer or if they enter an out-of-range number, they'll be required to try to enter a different number.

We're going to fix this in a slightly elaborate way. To start, create a new module called `Parser` in the `src` directory. Add this module to the `exposed-modules` section of the `library` in the `.cabal` file. The code we're adding will require you to add the `regex-applicative` dependency to the library. Fill in the module with this code:

```
module Parser where

import Data.Char
import Text.Regex.Applicative

parseIntRegex :: String -> Maybe Int
parseIntRegex = match intRegex

intRegex :: RE Char Int
intRegex = spaces *> parse14 <* spaces
  where
    spaces = many (psym isSpace)
    parse14 = (\c -> ord c - 48) <$>
      psym (\c -> isDigit c && c <= '4' && c > '0')
```

Now go to the `Runner` module, import your new `Parser` module, and swap out `parseIntRegex` for `readMaybe` in the `runQuestion` function. Rebuild your code and rerun the `run-quiz` executable to make sure it still works as intended. In fact, you should find that you'll get re-prompted if you enter an out-of-range number now.

Exposing and Hiding

In `app/RunQuiz.hs`, try importing the `Questions` module. You should find that the code doesn't compile, since this is a hidden module.

Remove that import, and then in the `.cabal` file, move `Runner` from the `exposed-modules` section to the `other-modules` section and rebuild. Once again, you should see a compilation error. Fix this once again by moving `Runner` back as an exposed module.

Lecture 6 - Test Suites

Render Tests Suite

To start, head to `Quiz.cabal` and examine the test suite called `render-tests`. See that its main file is in `test/RenderTests.hs` and that it depends on our library (`Quiz`) as well as the `tasty` and `tasty-hunit` libraries.

This is our only test suite so far. Try running it:

```
stack test
```

You should find that it doesn't compile because it depends on the `Questions` module, which should still be hidden. Fix this by exposing the `Questions` module in the `library`. Then run the tests again.

They'll run, but you'll see a test failure. In this particular case, the test case is actually wrong. So go into the `RenderTests` file and fix it so the expected output matches the actual output. Look around this file and get a feel for how the test cases work. Notice that it is given as `module Main` at the top, and it uses a `main :: IO ()` expression just like an executable.

Now when you run `stack test`, you should see the suite passes.

Making a New Suite

Now you'll make your own suite, called `parse-tests`. Its `main-is` file will be `test/ParseTests.hs`. Create this file, and implement a series of cases to validate the behavior of the `parseIntRegex` function from the last lecture (which should be in the `Parser` module). Here are some cases to check:

```
"1" -> Just 1
"  2  " -> Just 2
"34" -> Nothing
"-1 " -> Nothing
" A" -> Nothing
```

You can use the `stack test` command, and it will run both suites. Hopefully your new suite passes.

You can run your two suites in isolation by using these command:

```
stack build Quiz:test:parse-tests
stack build Quiz:test:render-tests
```

Alias Commands

Now make an alias for each of the two test commands. For example, you might use the alias `qpt` ("Quiz Parse Tests") to run the `parse-tests` suite, and `qrt` for the `render-tests` suite.

Follow the instructions in the walkthrough videos, depending on your platform (windows vs. Mac/Linux).

Run your alias commands to ensure they work!

Benchmarks

Finally, look at the `benchmark find-numbers` section in the `.cabal` file. You'll want to go to `benchmarks/FindNumbers.hs` to see its source code.

You can run the benchmark with either of two commands. The first of which would run **all** benchmarks if you had more than one. The second would only run this benchmark (Windows users pay attention to the walkthrough; you need to run the command `chcp 65001` or this command will fail when it tries to print the mu character `μ`).

```
stack bench
stack build Quiz:bench:find-numbers
```

Note that it will take a while to download the dependencies for the Criterion library!

Go to `FindNumbers.hs`. Add an additional benchmark using a size of 100000 elements. On the line that calls `generateInputs`, you'll need one more element in the input list, and you'll need to unwrap one more tuple in the resulting list. Run the benchmark again and see how long it takes (it will take a minute or two). Take a note of the average times for each size of suites.

Now switch the definition of `findNumbers` at the top of the file. Instead of using `findNumbersMap`, have it use `findNumbersList`. Run the benchmark again, but disable the largest benchmark first (it will take too long)! Take note of the average times. You should be able to observe that lists are not very efficient for the `lookup` operation!

Expand the benchmarks so that for each generated input, it calls both `findNumbersList` and `findNumbersMap` (except don't run the largest number with lists). This way, you can compare the results within the same benchmark run.

Lecture 7 - Extra Dependencies

Note: On Linux, this lecture requires you to have installed the `zlib1g-dev` package (`zlib1g` didn't work for me). Also, as with the benchmark code, these exercises do involve a lot of waiting around for new packages to load and compile.So for this lecture, it might be good to have something to do while you're waiting!

Adding an Extra Dependency

For this lecture, we'll be working in the `Setup-hs/Mail` directory. Try building the project in there with `stack build`. This project uses the LTS `17.9` Stack resolver, which corresponds to an older version of GHC than we've been using so far (`8.10.4`). So you might have to wait a bit while it downloads and installs this version.

You should find that the build ultimately fails. Our source code in `src/Mail.hs` depends on the Hailgun library (`Mail.Hailgun`), and we haven't included this dependency in the `.cabal` file. Try adding it to the library dependency section and build again.

The build won't succeed, because the Hailgun library isn't actually part of LTS `17.9`! You need to add it instead in the `extra-deps` section of the `stack.yaml` file. You'll need to explicitly specify the version of the package as well (`hailgun-0.5.1`). Now when you build your code (it will take a little while since `hailgun` has many dependencies), it should succeed!

Using a GitHub Dependency

Now let's suppose we want to upgrade this project to use GHC `9.0.2`. To do this, we'll change the resolver in the `stack.yaml` file to `19.24`. Try building.

You should find that Stack cannot create a successful build plan, because `hailgun-0.5.1` requires some conflicting dependencies. And unfortunately, there is not (at time of writing) a newer version of the Hailgun package on Hackage.

However, instead of using `hailgun-0.5.1` in the `extra-deps` section, try instead filling in this GitHub description:

```
extra-deps:
  - git: https://github.com/jhb563/hailgun.git
    commit: 41ac6bad08b67f320f8eeba9bf4abb94bcb0513c
```

This refers to a GitHub fork I made of the Hailgun package to make it compatible with GHC `9.0.2`. You can take a look at the necessary changes by going to this URL: https://github.com/jhb563/hailgun/commit/41ac6bad08b67f320f8eeba9bf4abb94bcb0513c.

After making this change, try building again. You should find that it works!

Lecture 8 - Haskell Language Server

Verify HLS Installation

There's not much to do in this lecture, since most of the configuration takes place within the context of the editor/IDE you're working with, and we'll consider a few examples in the coming lectures.

Just check that HLS is installed in a couple ways. First use `ghcup list` (or `ghcup tui`) and check that at least one of the versions for HLS is installed (two checkmarks next to it or else the green **IS** in Windows Powershell).

Then run the following command:

```
haskell-language-server-wrapper --version
```

This should show you the version number, as well as the path for the executable.

Changing HLS Versions

It's also important to know how to change versions though, because this affects which projects and GHC versions you can get support for! Run `ghcup list` once again, and this time take note of which GHC versions are labeled with **`hls-powered`** in green lettering to the right.

Now try installing a new version of HLS, for example, `1.8.0.0`, or `1.9.0.0`.

```
ghcup install hls 1.8.0.0
```

Installing a new version should automatically set it as the selected version. Run `ghcup list` once again, and observe that different GHC versions are now indicated as **`hls-powered`**! Each HLS version supports different GHC versions, so be cognizant of that.

Lecture 9 - VS Code Setup

For this lecture, just follow along with the instructions given in the walkthrough video to get VS Code working with Haskell! Here's a rough write-up of the instructions.

Install VS Code and Extensions

First, download and install VS Code: https://code.visualstudio.com/download.

Next, install the required extensions. In general, you should only need the "Haskell" extension (https://marketplace.visualstudio.com/items?itemName=haskell.haskell).

However, if you are using Windows Subsystem for Linux, you'll first need to get the "Remote - SSH" extension if it is not already installed (https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh).

On WSL, you'll need to ensure the Haskell extension is installed *on the WSL remote,* rather than locally. To do this, make sure you open up the `Quiz` project (next step) on WSL remote *before* installing the extension.

Open the Project

Now go ahead and open up the `Quiz` project in VS code! Just use `File -> Open Folder` and locate it in the directory finder. On WSL, you'll want to use `"Open Folder in WSL…"` rather than `"New WSL Window"`.

Configuration

Next, open your `settings.json` file (`Ctrl+Shift+P` and search for "Open User Settings (JSON)"), and add the following line:

```
{
  "haskell.manageHLS": "GHCup",
  ...
}
```

Our Quiz project uses Stack resolver LTS `20.8`, which requires GHC `9.2.5`, which requires HLS `1.9.1.0` or `1.9.0.0`. The walkthrough videos use `1.9.0.0`, but after those were recorded, `1.9.1.0` came out, and for VS Code you'll want to use the more recent version! So let's use GHCup to set these values.

```
ghcup set ghc 9.2.5
ghcup set hls 1.9.1.0
```

Now, restart your editor so that HLS can restart.

Try it Out!

You should now be able to open `Runner.hs`, see that it has proper syntax highlighting, and start meddling with the file to see the kinds of hints you'll get from HLS.

For example, you can put parentheses around the string `"You have finished all the questions!"`, and you should find that the editor provides a lint suggestion to remove the parentheses since they are redundant.

You could also change the type of `runQuiz` to `IO Int`, and it should display that this causes a compilation error.

Last, you can try replacing the call of `mapM_` in runQuestion with `forM_` (and reverse the order of the arguments). You should observe that an auto-complete hint is available as you start typing it in. You need to import this function (via `Control.Monad`). And it's quite easy to do this. You can move your cursor over the `forM_` expression, and VS Code should bring up a menu of options, and you can use a keyboard shortcut (like `Ctrl+.`) to automatically add the import at the top!

Note: there may be an initial delay (even up to 60 seconds) before the hints start working, especially if you just restarted HLS.

Lecture 10 - Vim Setup

Install Node

First, you need to install NodeJS, with a version that is at least `14.14`.

On Linux you need commands like these:

```
curl -sL install-node.vercel.app/lts > install-node.sh
sudo chmod +x ./install-node.sh
sudo ./install-node.sh
```

For Mac and Windows, you can go to https://nodejs.org/en/download and use the installers.

Install Vim-Plug

Now we need the Vim-Plug system for installing plugins. On Linux and Mac, we do this with a `curl` command:

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
  https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

On Windows, you'll use the equivalent command `iwr`, which has different arguments:

```
iwr -useb https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim |`
    ni $HOME/vimfiles/autoload/plug.vim -Force
```

Install CoC Plugin

Once Vim-Plug is installed, add this to your Vim Config (`.vimrc` or `_vimrc`) (remember to replace `.vim` with `vimfiles` on Windows).

```
call plug#begin('~/.vim/plugged')
Plug 'neoclide/coc.nvim', {'branch': 'release'}
call plug#end()
```

Then you'll need to re-source the Vim Config file (or else re-open Vim entirely), and then run the `:PlugInstall` command. You should see a vertical split panel open showing the progress and that it succeeds.

Configuration

Now you just need to configure the plugin! This is done with a file `coc-settings.json` in your vim folder (`~/.vim` or `~/vimfiles`). You should add the `"languageserver"` key with the following object:

```
{
  "languageserver": {
    "haskell": {
      "command": "haskell-language-server-wrapper",
      "args": ["--lsp"],
      "rootPatterns": ["*.cabal", "stack.yaml", "cabal.project",
                       "package.yaml", "hie.yaml"],
      "filetypes": ["haskell", "lhaskell"]
    }
  }
}
```

Try it Out!

Now you should be able to open up your `Quiz` project in Vim! Remember, from the command line, you should open files from the root of the project, rather than changing into the source directory.

```
cd Quiz
vim src/Runner.hs
```

The following will not work (it won't find imports properly):

```
cd Quiz/src
vim Runner.hs
```

When you open the `Runner` module, you can take the same steps as last lecture to demonstrate that code hints are working:

Open up `src/Runner.hs` and try putting parentheses around the string `"You have finished all the questions!"`. The editor should provide a lint suggestion to remove the parentheses since they are redundant.

Try changing the type of `runQuiz` to `IO Int`, and your editor should display that this causes a compilation error.

Last, try replacing the call of `mapM_` in runQuestion with `forM_` (and reverse the order of the arguments). An auto-complete hint should be available as you start typing it in.

As with VS code, you may still see a delay before the hints appear.

Lecture 11 - Emacs Setup

Install MELPA and LSP Mode

In order to enable Emacs integration with HLS, we have to install the `lsp-mode` and `lsp-haskell` libraries, as well as `company` for autocompletion. These aren't in the default package archive; you have to add MELPA (https://melpa.org/#/) to your archive list. So to begin, add these lines to the top of your `~/.emacs` file (or wherever you place the initialization).

```
(require 'package)
(add-to-list 'package-archives '("melpa" . "https://melpa.org/packages/") t)
(package-initialize)
```

Now open up your editor and run these emacs commands to install the packages. For those unfamiliar with Emacs command syntax, a command like `M-x` means "modifer+x", and the modifier key is usually mapped to "alt" by default, but it may also be the "escape" key. Then `RET` indicates you should hit Enter/Return before the next part.

```
M-x package-refresh-contents RET
M-x package-install RET lsp-mode RET
M-x package-install RET lsp-haskell RET
M-x package-install RET company RET
```

Customize for Haskell

Now go back into your `.emacs` file and add a few lines to customize LSP to work with Haskell.

```
(require 'lsp)
(require 'lsp-haskell)
(add-hook 'haskell-mode-hook #'lsp)
(add-hook 'haskell-literate-mode-hook #'lsp)
(customize-group 'lsp-haskell)
```

After adding these, you should ensure that you restart the Haskell Language Server before opening Emacs again!

Try it Out!

After those few steps, you should be good to go! As in the last couple lectures, you can try opening up `src/Runner.hs` and make a few modifications to see the Haskell features. When you open the file, you'll be prompted as to how to find the directory with project configuration file (Quiz.cabal). I always go with the capital **I** option of specifying the directory manually. Here are the changes you can try to see that your editor hints are working:

1. Add parentheses `"You have finished all the questions!"` to see a lint suggestion.
2. Change the type of `runQuiz` to `IO Int` and see the compilation error.
3. Try replacing `mapM_` with `forM_`, and you should see some autocomplete suggestions.

As before, there may be a delay before hints start appearing.

Now you're done with all the exercises! You can watch the Lecture 12 Conclusion video for a summary of what we've done and for some suggestions on next steps.