Module 1 Exercises

Monday Morning Haskell
Practical Haskell Course

Lecture 1

Throughout this course, you'll be doing exercises by pulling code from Github and running it with either GHCI or the Haskell Stack tool.

For this lecture, your first job is to make sure you have Haskell (and Stack) running on your machine. If you don't, you can install them quite easily. On Mac and Linux, you should be able to run:

```
>> curl -sSL https://get.haskellstack.org/ | sh
```

For Windows, you should download the appropriate installer from the documentation. You'll know you've succeeded when running the `stack` command at the terminal gives a detailed usage error, instead of a "`command not found`" error.

Next, you'll want to clone the Github repository for this class. You should already have a Github account that has access to the repository.

```
>> git clone https://github.com/jhb563/PracticalHaskell.git
```

If cloning the repository fails because of permission errors, send me a message on the Slack group or email me at `james@mondaymorninghaskell.me` ASAP!

Now you should checkout the branch for this module and pull it.

```
>> git fetch origin module-1
>> git checkout module-1
```

Lastly, try the following commands. If you already have Postgresql installed, these should work. If not (you see an error on `postgresql-libpq`), run these in the Lecture 2 exercises after installing:

```
>> stack setup
>> stack build
>> stack ghci
```

If these are successful, the last should bring up the GHCI REPL where we'll do a lot of our early programming. It'll generally give you a message like "`The main module to load is ambiguous...You can specify which one to pick`". When this happens, you can always enter `0`! We don't need to access and `main` functions. If that's all set up you're good to go!

As a final note, you'll typically see that each set of exercises has a particular test suite attached to them. The first way to run a particular test suite (say, `lecture-2-tests`) is with the full stack build command as follows:

```
stack build PracticalHaskell:test:lecture-2-tests
```

However, if you run the `stack install` command on the `module-1` branch, this will create some proxy commands to do this for you. You'll subsequently be able to run the previous command like so:

```
test-2
```

And so on, for each lecture number.

Lecture 2

For these exercises, you should first make sure you've installed PostgreSQL and SQLite on your local machine. Watch the two screen-casts on the course page to help you out with this. The screen-casts show me installing both programs on my Windows machine. If you're working with Mac or Linux, the exact steps will be a little different, but the same ideas will apply. You should now definitely be able to run stack commands to get the repository building:

```
>> stack setup
>> stack build
>> stack ghci
```

There's one thing in the Postgres screencast I want to correct. You MUST use the username `postgres` and password `postgres` for your `mmh` database! Don't customize this. The tests depend on it!

When you've done this, you should have a local version of Postgres running in the background. Using the `CREATE DATABASE` command, make a new database within Postgres called `mmh`. Then connect to that database with the `psql` command `\c mmh`. (It's a little annoying, but you'll have to do this every time you re-open `psql`). You should also create an SQLite database in your version of the course Github repository. Go to the `local-dbs` folder and create the database file `lecture-1-2.db`.

Then, after having watched the lecture slides, create a `users` table, both in your `lecture-1-2` Postgres database as well as in your SQLite database. This table should match the format presented in the slides. Insert a few users with the following information. You can use whatever numbers you like for the ID keys (except 99, used by the tests):

```
'Chris' | 'chris@test.com' | 32
'Kathleen' | 'kathleen@corp.com' | 28
'Michael' | 'michael@college.edu' | 21
```

Now create a new table `articles` in each database. Each article will have three fields. The first will be the `title` of the article, a text field. The second will be the `body` (also text), and the third will be a date field `publishedAt` (use `timestamptz` for the type). The table should also use an integer as its (non-null) primary key. Insert 3 articles into each database with the following titles:

```
'Introduction to Haskell'

'Databases in Haskell'

'Production Quality Systems'
```

You can use whatever information you want for the body and published at times. Note that for a datetime field, you'll want to supply a string with the time format `'YYYY-MM-DDTHH-mm-ssZ'`. For example, `'2018-12-15T10:30:00Z'`. NOTE!!! If you omit the `'Z'` character, it will cause test failures for strange Haskell library reasons.

As a reminder, do this in both your Postgres database AND your SQLite database. Run the `lecture-2` test suite in order to verify that these works. The tests will insert some new items into your database, but then delete them.

Lecture 3

For this module we'll be working strictly with our Postgres database. We'll be generating new tables for our `users` and `articles` in these exercises. You should either delete these tables like so:

```
DROP TABLE users;
DROP TABLE articles;
```

Or you can rename them:

```
ALTER TABLE users RENAME TO users_old;
ALTER TABLE articles RENAME TO articles_old;
```

Now take a quick look at the existing code in `Schema.hs` and `Database.hs`. The `Schema` has all the code for the `users` table we went over in the lecture slides and the screencasts. The `Database` file has our code for migrating. All you need to pay attention to is the `migrateDB` function, which is exported to the `migrate-db` executable.

Assuming you've run `stack install`, you can now run the command `migrate-db`, and it will create our users table! Run the command and verify this by connecting to your database and using `\d users` to describe the table.

Now it's your turn! Following the example with `users`, create our `articles` table, which should have the same fields we had in the previous lecture. Use `UTCTime` for the publish time field type and don't forget to derive basic typeclasses!

When you're done, we'd like to be able to `migrate-db` again. But the executable on your local path will still use the old version. You can either run `stack install` again, or you can just run `stack build` and use `stack exec migrate-db`. Either of these will create the `articles` table in your database!

Test your code with the `test-3` command!

Lecture 4

To start off this lecture, uncomment the lines for the `Lecture4`, `Lecture5`, and `Lecture6` modules in `PracticalHaskell.cabal` (look around lines 25-27). The modules wouldn't compile until now because they reference the `Article` type you just wrote! Leave the other modules commented for now.

Next, a few of the upcoming exercises depend on loading the file `database_1.sql` into your Postgres database. To do this, first delete all the entries in your users and article tables by running the following commands from your `psql` prompt:

```
delete from articles;
delete from users;
```

Then, run this command from the root of the course directory.

```
psql -U postgres mmh < db_dumps/database_1.sql
```

You might get an error that `multiple primary keys...are not allowed`. You can ignore it. Just verify that each table has 100 like so:

```
select count(*) from users;
select count(*) from articles;
```

Now, take a look at `Lecture4.hs`.

To start with, define a simple user, `myUser`, and an article, `myArticle`. The user should have the name `"Chris"`, and the article should be titled `"Intro to Haskell"`. You can fill in the other fields however you like. Use record syntax to familiarize yourself more with how the field names work.

Next, make these into entities, `myUserEntity` and `myArticleEntity`. Give them both a key of 1. We'll see in the next lecture how database keys are automatically assigned upon insertion.

Finally, take a look at the simple function, `fetchAllUsers`, that already exists to fetch all the users in our database. We'll learn more about the implementation of this function in the next lecture. For now, use this function to fill in the definition of `printAllKeys`, which should print out the database keys of every user in our database. This function takes a file `Handle` as an input. Print to this handle using `hPrint`, rather than standard out.

Lecture 5

All these exercises are in `Lecture5.hs`. This lecture also includes automated tests!

To start, let's get used to insertion and deletion. Fill in the function `insertAndPrintKey`. This function should first insert `myUser` into the database. Use the `runAction` function from `Database.hs` to convert your Persistent query into an IO action. Then, print the key of the resulting entity. Load this module into GHCI and run the function. You should be able to bring up a separate terminal window, connect to the database, and run a query to see that we have a user by the name of `"Lecture 5 User"` with the ID you just saw.

Now that you know the database ID of that user, fill in the `undefined` value of the `deleteNewUser` function. Re-compile the module, and run that function. You should now find the user is deleted when you query the database in your other terminal.

Next, let's fill in some more basic SQL functions. In `fetch100`, write a function to retrieve the article with the ID 100. Return its title and published date. Follow the guidelines in the function, but note that you'll need to fill in the type signature for `fetchQuery`! You can use the `fromJust` function to get around any headaches caused by using a `Maybe` value!

Next, fill in `lastYearsArticles`. This query should return all the articles that were published in the year 2018. Order them by title. (Hint, use `ModifiedJulianDay` and this link).

Finally, fill in `getYoungUsers`. This should return all users under the age of 23 (not inclusive). However, you should limit your search to 10 users, and order them *backwards* alphabetically by name.

Lecture 6

Take a look at `Lecture6.hs`. Instead of the `runAction` function we've been using from `Database.hs`, it's a slightly different version. We can see from running it that it logs a lot more information than we need! If you open up GHCI, load the `Lecture6` module and run `fetchSpecialUsers`, you'll see it logs the whole query we write!

To fix this, we're going to write a filter for our logger using compile driven development. Then we'll apply it in the `runAction` function. First use Hackage/Hoogle to find the `filterLogger` function. This should tell you the proper type to use for the `logFilter` expression. Fill this in as `undefined` for now.

Next, determine how to use `filterLogger` within the `runAction` expression to filter out the messages. Its type will tell you where it goes! Make this edit and ensure your code compiles BEFORE filling in the `logFilter`.

Finally, use the type of `logFilter` to fill in the definition. You can ignore the first argument, and then expand out the second using a pattern match. We should only print messages that have severity of `Info`, `Warn`, or `Error`. Again, refer to the documentation for help on any relevant types.

If you get stuck you can, of course, check the definition we have in `Database.hs` that will show you the answer. Verify your solution works by running the query again and ensuring there is less output (it shouldn't log the actual SQL query).

To conclude this lecture, write your most complicated query yet. Write a multi-part query `getSpecialPairs` to find all pairs of users and articles where the first letter of the user's name matches the first letter of the article's title. Don't worry about efficiency! The proper way to do this would be with a join query, but we don't know how to do that yet! So just use multiple queries on single tables. Use compile driven development like we showed in the screen-cast. Your answers should be sorted by ID of the user.

Lecture 7

We're going to be changing our schema a bit in this lecture, so there are some setup tasks involved.

Your first task is to add the `authorId` field to the `Article` type in our schema. This should mimic what we had in the lecture slides, where the type is `UserId`.

Next, you should add a new type to our schema. This type should be called `Comment`, and it should reflect a comment a user has made on our article. This type should have TWO foreign keys, one for the user making the comment, and the second for the article they are commenting on. Name the first field `userId` and the second field `articleId`. Each comment should also have a `Text` field (`body`) for what the user actually wrote, as well as a `submittedAt UTCTime` field. Follow the other conventions we have for our types as well. The table name should be `comments`, and you should derive all the same type classes.

Now, because our schema has changed, a lot of the old code you wrote won't compile anymore. Go to `PracticalHaskell.cabal`. So you should re-comment the `Lecture4`, `Lecture5`, and `Lecture6` modules. Then you should un-comment `Lecture7`, `Lecture8`, and `Lecture9` (leave `Lecture10` and `Lecture11` commented out for now). Build your code to make sure everything works.

Once you're done with that, we need to modify our database to stay up to date with that! We'll get to more sophisticated versioning processes at the end of this module. But for now, you should once again connect to the database and run:

```
drop tables articles;
drop table users;
```

This time, do *not* use the migration executable. Instead, run the following command to get a new dump of database items like we had in lecture 4. This will actually generate the tables for you.

```
psql -U postgres mmh < db_dumps/database_2.sql
```

To confirm the migration has worked, you can run `migrate-db`. It shouldn't have to run any commands.

Now for a couple more tasks. In `Lecture7.hs`, fill in the definition of `getCommentsFromUser`. This should take a user ID and return all the comments a user has made. The comments should be ordered by article (smallest ID first), and then by the comments' dates, with the most recent first.

Now, fill in `getCommentsOnUser`. This should take a user ID and retrieve all the comment entities on all the articles a user has written. This time, just order by the comments' dates (put the oldest first). As before, these will be multi-part queries with Persistent.

Lecture 8

For these exercises, we're just going to re-write a number of the queries we've written in past lectures. Only now, we're going to do it with Esqueleto! All your code for this section will go in `Lecture8.hs`. Just fill in all the missing function definitions, using joins whenever possible! As a reminder, here's what the queries mean:

`lastYearsArticles`: Return all the articles that were published in the year 2018. Order them by title.

`getYoungUsers`: Return all users under the age of 23 (not inclusive). However, you should limit your search to 10 users, and order them *backwards* alphabetically by name.

`getSpecialPairs`: Find all pairs of users and articles where the first letter of the user's name matches the first letter of the article's title, except do this only when that first letter is `'E'` or `'T'`. This last modification spares you from having to post-process any results like we had to in Persistent. Look in the Esqueleto documentation for the functions `like` and `(%)` for some help.

`getCommentsFromUser`: Take a user's ID and return all that user's comments. The comments should be ordered by article (smallest ID first), and then by their date, with the most recent first.

`getCommentsOnUser`: Take a user ID and retrieve all the comment entities on all the articles a user has written. Sort by the time the comments were submitted, starting with the oldest.

(Hint: You can use an `InnerJoin` expression as one of the arguments to another `InnerJoin` expression. You'll need 2 `on` statements, but I'll leave it to you to figure out what the right order is for them!).

Lecture 9

For this lecture, you'll start by modifying `Schema.hs`, since that's where our types live.

First, make `ToJSON` and `FromJSON` instances for all three of our schema types. Follow the general example from the lecture slides. Use field names that are all lower camel case and match our schema field names. Note the tests won't compile until you create all these definitions.

Run the tests (with the `test-9` command). Once the first set of tests passes (3 sections with `JSON Basic` in the title), you should move onto the next part.

Now comment out your instance definitions. Create new instances using the deriving method shown in the lecture slides.

The tests shouldn't pass, since the derived instances are different from what we had! You can see what they look like in the test failures, or you can load up GHCI and use the `Data.Aeson.encode` function on some of the sample objects in the `Lecture9` module.

To fix this, you'll want to modify the `defaultOptions` object you pass as an argument to `deriveJSON`. You'll mainly want to change the `fieldLabelModifier` on those options. (Hint, use `dropXAndLowerFirst`, imported from `Utils`!)

Now, we modified the `Schema` file directly because you should only define instances for a typeclass either in the module where the type is defined or where the typeclass is defined. In fact, if you do otherwise, you'll typically generates a compiler warning.

But there is a workaround for this. Open up `Lecture9.hs`, where we've defined `newtype` wrappers for all our types. Since the `newtypes` are defined in this module, it's totally fine to make instances for those `newtypes` in this module, rather than the `Schema` where the original types are defined. This workaround also let's you effectively define different instances for the same type!

Fill in the `ToJSON` and `FromJSON` instances for the `newtype` wrappers, but make the instances match the default derived instances we got from `Data.Aeson.TH` (fill them in manually though)! Once you're done with this, all the tests should pass! Observe how we don't need to encode anything about the `newtype` wrapper in the instance if we don't want to!

Lecture 10

Start off this lecture by uncommenting `Lecture10` and `Lecture11` in `PracticalHaskell.cabal`!

In this lecture we'll explore some of these alternative type concepts by adding one more type to `Schema.hs`. We'll call this type `ArticleReaction`. This will allow users to `Like`, `Love`, or `Dislike` one of our articles just as they would on Facebook. It should have four fields.

First, it must have an `articleId`, just as comments do. Second, it will have a field `userId`, for the user that reacted. However, this field should be nullable, since users won't have to be logged in to react to an article. Third, it should have the `ReactionType` (call the field `type`). You can find this type in `SchemaTypes.hs`.

Finally, you should add one more type to `SchemaTypes.hs`. The type should be called `Metadata`, which should have four fields, `reactionTime`, `previousLikes`, `previousLoves`, and `previousDislikes` (Just use `Int` for these last three fields).The last field of `ArticleReaction` should be called `metadata`, and use this new type.

Once you've added these, try building your code. You should find that there are errors because we haven't made these types implement the `PersistField` typeclasses. Do this by going into `SchemaTypes.hs` and add the relevant `derivePersistField` lines. Remember to derive `Show` and `Read` instances! The only “test” (by running the `test-10` command) for this lecture simply verifies that you have created this type and these instances correctly.

Once your code builds, use `stack exec migrate-db` to add the table to your database!

In `Lecture10.hs`, take a look at the `addReaction` function. Open up GHCI and run this function, which should insert a new `ArticleReaction` and print out the key. Now run `fetchNewReaction`, using the key as the argument. Observe that you can retrieve the reaction we inserted.

Now go back to `SchemaTypes`. Change the derivations so they use `derivePersistFieldJSON` instead of `derivePersistField`. You'll need to also derive the JSON instances. Now re-compile your code and try to run `fetchNewReaction` again. It should fail, because we've changed the schema and how we expect to de-serialize the data!

Save what the item looks like in your database right now (copy it down somewhere), and then delete it. Go through the process of adding and fetching again, and compare what the new serialization looks like by comparison.

For the last part of these exercises, let's explore uniqueness constraints. In `Lecture10`, observe the function `makeClone`. This will find a user, any user, in the database, and insert a new user with the same name, email and age. It will also print the key of the new so that you can use it in the `deleteNewUser` function. Run this function via GHCI, and observe that it works! Then delete the new user with the `deleteNewUser` function.

Now add a uniqueness constraint on the user's name AND email in the schema definition. Then re-compile your code, and run the `migrate-db` script again with `stack exec`. Now try running `makeClone`, and you should see that it fails! You could now try inserting a user with the same name, but NOT the same email (or vice versa), and observe that it still works!

Lecture 11

For the final lecture, we're going to write a couple migrations for our database, going beyond the simple automated migration we get from Persistent. There are no automated tests for this lecture. Start off by re-commenting `Lecture9` in the `.cabal` file, because it contains a reference to the old definition of `Comment`.

To start, let's do a migration we can handle automatically. In `Schema.hs`, let's add a column calling `rating` to the `comments` table. This will include an integer rating, which we can imagine as 1-5. We'll provide a default value of 3. Do this, run the automatic migration, and you should find that all the comments in the database use the value 3 for their rating.

Now go into the schema and remove the column. Try re-building and migrating again. The automatic migration will fail, since it's a lossy action! In `Lecture11.hs`, fill in `removeRatingOp` with an `Operation` to remove this `rating` column. Then you can load the module in GHCI and use `runRemoveMigration` to remove the column! Verify that the column is no longer present.

Now, re-do the process of adding the `rating` to the `Schema`. Ensure that the column is present and that everything has a rating of 3 again. We're now going to change the scale so that it's 1-10 instead of 1-5. For this migration, you should take the value from the existing column and double it. Fill in the operation, `doubleRatingOp`, and the migration expression, `doubleRatingMigration`. Note your versioning numbers should be different from your previous migration! Then run the `runDoubleMigration` expression from GHCI to run it. Verify that all the ratings are now 6.

As a general gotcha, if you run a migration and it doesn't work for some reason and you need to change the code, you have must either change the version number in the Migration element, or you must delete the entry in the persistent_migration table corresponding to the failed migration. If an entry exists with the same version as the second number in your Migration, it won't run again.

Now some “migrations” we can perform are sneaky, and Persistent won't be able to pick up on them on its own. Let's consider a scenario where we add a new type of reaction. Add `Surprised` as a new field on the `ReactionType` enumeration. Use the `insertSurprised` expression at the bottom to insert a new item into this table with the `Surprised` reaction type (fill in the second `undefined` expression with `Surprised`). Pick any article in the `articles` table to use as the parent, and fill in the `undefined` value with its key. Make a note of the key of this new reaction you've made.

Now remove the `Surprised` option from the enum (comment out the `insertSurprised` expression) and re-build your code again. Run the automatic migrations again. Notice that nothing happens! Now fill in the key of your new reaction in the `fetchSurprised` expression and run it from GHCI. Notice you get a run-time error, because we can't read the field with `Surprised`!

To fix this, write another `rawSql` operation in `surprisedToLikeOp` and `surprisedToLikeMigration`. Use these to change all values with the `Surprised` reaction type to `Like`. Then run your migration, and fetch the item again, verifying that it succeeds. (Gotcha: you have to include single AND double quotes in the SQL query string: `'\"Surprised\"'`).

Notice how we had a very hard process to remove an enum field! Worse, there were no safeguards on that! There are other ways our there to do this kind of migration, this one isn't necessarily the best! But hopefully this exercise gives you some ideas of the kinds of headaches you can encounter with migrations!