Using optparse-applicative

What are we doing here?

We're going to make a command-line todo list application. This meetup focuses on understanding and using the library `optparse-applicative` and writing the command-line argument parsing portion of that application.

Why `optparse-applicative`?

- well maintained and documented
- lots of blog posts to use to get familiar
- seems fairly “batteries included”
- covers many use cases

Remember `Applicative` and `Alternative`

- the `Applicative` pattern proves useful for parsing command line arguments!

  ```
  DataConstructor <$> argument <*> argument
  ```

- and using `Alternative` lets us parse in parallel! This will only fail if *both* parses fail.

  ```
  twoTries :: String -> Maybe Char
  twoTries xs = upper xs <|> numbr xs
  ```

- This turns out to be a great combination for reading command line arguments.

Let's look at a sample program

- Check out ./app/optex.hs
- $ stack runghc ./app/optex.hs -o optex
- $ ./optex –help
- $ ./optex “julie”
- $ ./optex “julie” -e

OK, so onto our todo list!

We want to be able to - create new tasks;
- list the tasks, possibly one or many at a time; - update tasks; - delete tasks.
So those will be our command line arguments.

Data

We'll start by writing an appropriate datatype.
We can use a sum of products for this!

```
data Command = New String | ...
```

Parsers!

Next we want a parser for each data constructor.

```
parserNew :: Parser Command
parserNew = New <$> strArgument (metavar "TASK_NAME")
```

`strArgument` is a builder for `String` arguments. *Builders* let you *build* options, add modifiers to those options, and then combine your parsers into a single (parallel) parser.

and subparsers!

```
subparser :: Mod CommandFields a -> Parser a
```

Builder for a command parser. The `command` modifier can be used to specify individual commands.
`subparser` (see library) allows us to read commands

```
command :: String -> ParserInfo a -> Mod CommandFields a
```

`command` adds a command to a subparser option.