Тема 9. Упражнения

Упражнение 9.1 (Эхо)**.** Реализуйте следующие варианты программы “эхо”:

1. `echo :: IO ()` — программа, печатающая на экран построчно то же, что ввёл пользователь;
2. `shoutEcho :: IO ()` — программа, печатающая на экран построчно то же, что ввёл пользователь, но заменяя все буквы на заглавные:

```
>>> shoutEcho
Привет, мир!
ПРИВЕТ, МИР!
```

3. `echoByWord :: IO ()` — программа, печатающая на экран построчно то же, что ввёл пользователь, но заменяя каждое слово на предложение и игнорируя изначальную пунктуацию и небуквенные символы:

```
>>> echoByWord
Привет, мир!
Привет. Мир.
Я хочу 40 килограмм хурмы.
Я. Хочу. Килограмм. Хурмы.
```

Упражнение 9.2 (Комбинаторы для ввода-вывода)**.** Реализуйте следующие функции:

1. `confirmIO :: String -> (Bool -> String) -> IO Bool` — запросить у пользователя подтверждение:

```
>>> confirmIO "Да или нет?" (\b -> if b then "да" else "нет")
```

```
Да или нет? да
True
```

2. `continueIO :: IO a -> IO ()` — запустить программу, пока пользователь не попросит остановить:

```
>>> continueIO (putStrLn "Привет!")
```

```
Привет!
Продолжить? [да/нет] да
Привет!
Продолжить? [да/нет] да
Привет!
Продолжить? [да/нет] нет
```

3. `verboseIO :: Mode -> IO () -> IO ()` — запустить программу, только если установлен режим `Verbose` или более подробный (`Debug`):

```
data Mode = Debug  Verbose  Normal | Silent deriving (Show, Eq, Ord)
```

4. `maybeIO :: Maybe (IO a) -> IO (Maybe a)` — запустить программу, если она есть;

5. `sequenceResultIO :: [IO (Result a)] -> IO [a]` — запустить последовательность программ и собрать все успешные результаты;

```
data Result a = Success a | Failure String
```

6. `unfoldIO :: (a -> IO (Step a)) -> a -> IO [a]` — начиная с начального значения типа `a`, продолжайте применять функцию к текущему значению, чтобы получить следующее (`Next`), пока возможно (до получения `Stop`); все результаты, включая начальное значение, собираются в итоговый список.

```
data Step a = Stop | Next a
```

7. `forStateIO :: s -> [a] -> (s -> a -> IO s) -> IO s` — начиная с некоего начального состояния типа `s`, обработайте элементы списка типа `[a]` слева направо, применяя функцию типа `s -> a -> IO s` на каждой итерации, чтобы обновить текущее состояние; программа должна вернуть итоговое состояние типа `s`:

```
printConsLength :: Int -> String -> IO Int
printConsLength totalLength s = do
  let n = length s
      newTotalLength = totalLength + n
  putStrLn ("добавляем " ++ show s
    ++ " (длина = " ++ show newTotalLength ++ ")")
  return newTotalLength
```

```
>>> forStateIO 0 ["привет", "мир", "!"] printConsLength
```

```
добавляем "привет" (длина = 6)
добавляем "мир" (длина = 3)
добавляем "!" (длина = 1)
10
```

8. Реализуйте **полиморфную** функцию высшего порядка `iforIO`, которая выполняет тело цикла на каждом элементе и его индексе в заданном списке, и собирает результаты в список.

```
example = do
  iforIO [1, 2] (\i n ->
    iforIO "ab" (\j c -> do
      print ((i, j), replicate n c)
      return [(n, c)]))
```

```
>>> example
```

```
((0,0),"a")
((0,1),"b")
((1,0),"aa")
((1,1),"bb")
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Упражнение 9.3 (Трекер задач)**.** Завершите реализацию трекера задач из лекции и расширьте её поддержкой следующих функций:

1. сохранение и загрузка состояния приложения из файла по запросу пользователя;
2. автоматическое сохранение и загрузка состояния приложения из файла (при выходе и старте приложения);
3. отмена предыдущего действия по запросу пользователя (если возможно).