Weekly Assignments 4

To be submitted: Friday, 9 February 2018

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious. Please also pay attention to stylistic issues. The goal is always to submit code that does not just correctly do what was asked for, but also could be committed without further changes to an imaginary company codebase.

W4.1 Packaging

Prepare a Cabal package to contain all your solutions so that it can easily be built using cabal-install or stack.

Note that one package can contain one library (with arbitrarily many modules) and possibly several executables and test suites.

Please include a `README` file in the end explaining clearly where within the package the solutions to the individual subtasks are located.

Please do *NOT* try to upload your package to Hackage.

W4.2 Nested datatypes

Recall the type of perfect trees,

```haskell
data Perfect a = Z a | S (Perfect (a, a))
```

Subtask 4.2.1

Define a function

```haskell
reverse :: Perfect a -> Perfect a
```

that creates a perfect tree of the same shape, but with all the leaves in reversed order.

Subtask 4.2.2

Define a function

```
index :: Perfect a -> Int -> Maybe a
```

that looks up the element with the given index (from the left, starting at `0`), and fails if the index is out of bounds. The function should have logarithmic complexity on the number of elements in the tree.

Subtask 4.2.3

Define a function

```
build :: Int -> [a] -> Perfect a
```

such that `build n xs` builds a tree of height `n` (i.e., with `2 ^ n` elements), taking the leaves from the list (from left to right). The function is allowed to crash if the list is too short. Use the `State` monad.

Note: If this is too difficult, at least try to define the much simpler function

```
build :: Int -> Perfect ()
```

that builds a perfect tree of the given height with unit values in the leaves.

W4.3 Benchmarking

Have a look at the `criterion` package for benchmarking. Using criterion, test the efficiency of indexing the first, the middle and the last element of a perfect tree of different height. See how the runtime evolves with growing trees.

Try to pay attention to the following:

- the time for building the tree should not be measured,
- the time for indexing should not be cached due to laziness.

Turn the benchmark into a benchmark suite in the Cabal file so that it can be run via `cabal bench` or `stack bench`. Try to generate a nice html-report as well.

W4.4 Mini private keys

The Bitcoin Wiki at

https://en.bitcoin.it/wiki/Mini_private_key_format

describes a format for mini private keys.

You are allowed to use the `cryptonite` and `base58-bytestring` packages.

Subtask 4.4.1

Define a validator for mini private keys that determines if a given input is a valid mini private key and outputs the corresponding full private key.

This validator should be available as a library function and as an executable that reads the mini private key to test from the standard input.

Subtask 4.4.2

Define a function that transforms a private key into Wallet Import Format as described on

https://en.bitcoin.it/wiki/Wallet_import_format

Also implement the reverse transformation. This functionality should be available both as a library function and as an exectuable.

P2 Bitcoin network version handshake

This is another team exercise. It does not directly build on top of P1, although some concept may reoccur, and some code components might be reusable.

This task has a preliminary deadline of Tuesday, 13 February. At that day, you have to demonstrate the solution or at least provide a status report.

Nodes in the bitcoin network start their communication by exchanging a `version` message that is replied to by a `verack` message.

The goal of this assignment is to write Haskell code that can send a correct `version` message and receive a corresponding `verack` message from a bitcoin node running either on the mainnet or the testnet of the bitcoin network.

The format of these messages is binary and described here

https://en.bitcoin.it/wiki/Protocol_documentation#version

and here

https://en.bitcoin.it/wiki/Protocol_documentation#verack

The primary goal of this assignment is to define Haskell datatypes representing `version` and `verack` messages and to use the `binary` package to define suitable binary encoders and decoders for such messages that transform between values of the Haskell datatypes and the binary format.

Note that the modules `Data.Int` and `Data.Word` contain datatypes for fixed-size signed and unsigned integers. The `binary` package contains functions to encode and decode integers of various size in both big-endian and little-endian format. Use the `time` package for dealing with the representation of times.

Then compose a suitable `version` message as a Haskell value. Note that you can identify yourself as a node providing no services (for the services field).

Then figure out independently the address of a bitcoin node running on the bitcoin network and try to send your `version` message to that and see if you manage to get a reply. Note that testnet nodes use different magic values than mainnet nodes, so you have to send different messages depending on the network.

As a bonus and further evidence of having established a successful connection, you can try to get more information from the contacted node via sending e.g. a `getaddr` or `getheaders` message.