JPP 2021/22 — Program zaliczeniowy (Haskell)

Build Your Own Blockchain

Uwaga: ten opis jest długi, ale proszę się nie bać — rozwiązanie może być od niego krótsze.

A. Drzewa skrótów

Drzewo skrótów (Merkle tree), są strukturą danych, która umożliwia zwięzłe reprezentowanie potencjalnie dużych ilości danych za pomocą ich skrótów, w sposób umożliwiający niepodważalne wykazanie, że pewien element należy do drzewa o danym skrócie.

Popularny ich wariant (stosowany np. przez Bitcoin), który rozważamy w tym zadaniu, to zrównoważone drzewo binarne, w którym liście zawierają dane, natomiast każdy wierzchołek wewnętrzny zawiera skrót skrótów swoich potomków. Wierzchołki, które mają jednego potomka, dla celów obliczania skrótu traktujemy tak jakby miały dwóch potomków o identycznym skrócie.

Algorytm budowy drzewa skrótów dla niepustej listy elementów można opisać następująco (patrz Bitcoin wiki):

0. Jeśli lista ma długość 1, to skrót jej jedynego elementu jest korzeniem drzewa.
1. Pogrupuj elementy w pary (jeśli lista ma długość nieparzystą, ostatni element zduplikuj)
2. Dla każdej pary oblicz jej skrót
3. Wynikiem jest lista skrótów
4. Powtarzaj proces tak długo aż zostanie jeden skrót (korzeń drzewa)

W praktycznych zastosowaniach jako funkcji skrótu należy użyć funkcji bezpiecznej kryptograficznie (np SHA256, SHA3). W tym zadaniu dla uproszczenia użyjemy 32-bitowej funkcji skrótu (która zdecydowanie nie jest bezpieczna), zaimplementowanej w dostarczonym module `Hashable32`.

Stwórz moduł `HashTree` realizujący drzewa skrótu, spełniający poniższe warunki.

Moduł powinien dostarczać co najmniej operacji:

```
leaf :: Hashable a => a -> Tree a
twig :: Hashable a => Tree a -> Tree a
node :: Hashable a =>  Tree a -> Tree a -> Tree a
buildTree :: Hashable a => [a] -> Tree a
treeHash :: Tree a -> Hash
drawTree :: Show a => Tree a -> String
```

Funkcje `leaf`, `twig` i `node` budują węzły drzewa o odpowiednio 0,1 i 2 potomkach.

`buildTree` ma budować drzewo skrótów zawierające wszystkie elementy listy będącej jego argumentami (można założyć, ze lista ta jest niepusta).

`drawTree` ma być funkcją produkującą czytelną reprezentacje tekstową drzewa tak, aby:

```
>>> putStr $ drawTree $ buildTree "fubar"
0x2e1cc0e4 -
  0xfbfe18ac -
    0x6600a107 -
      0x00000066 'f'
      0x00000075 'u'
    0x62009aa7 -
      0x00000062 'b'
      0x00000061 'a'
  0xd11bea20 +
    0x7200b3e8 +
      0x00000072 'r'
```

(`+` i `-` oznaczają odpowiednio wierzchołek o jednym i dwóch potomkach); do wypisywania skrótów mozna użyć funkcji `showHash` z modułu `Hashable32`.

B. Dowody

Dowodem na przynależność elementu do drzewa o określonym korzeniu jest ścieżka (od korzenia do liścia), której każdy element zawiera informację, który potomek zawiera interesujący nas element oraz skrót drugiego z potomków.

```
type MerklePath = [Either Hash Hash]
data MerkleProof a = MerkleProof a MerklePath
```

Jeżeli wierzchołek ma jednego potomka, to można przyjąć, że jest to lewy potomek.

Zdefiniuj (w module `HashTree`) funkcje

```
buildProof :: Hashable a => a -> Tree a -> Maybe (MerkleProof a)
merklePaths :: Hashable a => a -> Tree a -> [MerklePath]
```

oraz instancję klasy `Show` dla `MerkleProof` i potrzebne funkcje tak, aby

```
>>> map showMerklePath  $ merklePaths 'i' $ buildTree "bitcoin"
["<0x5214666a<0x7400b6ff>0x00000062",">0x69f4387c<0x6e00ad98>0x0000006f"]

>>> buildProof 'i' $ buildTree "bitcoin"
Just (MerkleProof 'i' <0x5214666a<0x7400b6ff>0x00000062)

>>> buildProof 'e' $ buildTree "bitcoin"
Nothing
```

Zdefiniuj funkcję sprawdzającą dowód przynależności do drzewa skrótu o danym korzeniu:

```
verifyProof :: Hashable a => Hash -> MerkleProof a -> Bool
```

```
>>> let t = buildTree "bitcoin"
>>> let proof = buildProof 'i' t
>>> verifyProof (treeHash t) <$> proof
Just True
>>> verifyProof 0xbada55bb <$> proof
Just False
```

C. Blockchain

Stworzymy uproszczony łańcuch bloków (blockchain). oparty o schemat "Proof of Work". Uzupełnij dostarczony moduł `Blockchain` zgodnie z poniższym opisem.

Blockchain to łańcuch bloków transakcji z mechanizmami zapewniania jego rzetelności. W schemacie "Proof of Work" gwarancja rzetelności opiera się na trudności obliczeniowej generowania kolejnych bloków (zatem zmiana historii po upływie kilku bloków wymaga mocy obliczeniowej, której nikt nie posiada). Jako nagrodę za wykonywanie tej pracy obliczeniowej, ktokolwiek stworzy blok (co wymaga znalezienia wartości uzupełniającej blok tak, aby jego skrót spełniał określone warunki), otrzymuje w nagrodę pewną ilość monet (dla Bitcoina w tym momencie 6.25BTC, u nas 50 monet). Nagroda ta zapisywana jest w specjalnej transakcji wewnątrz bloku, zwanej 'coinbase'.

Cytując wiki Bitcoin:

"New bitcoins are generated by the network through the process of "mining". In a process that is similar to a continuous raffle draw, mining nodes on the network are awarded bitcoins each time they find the solution to a certain mathematical problem (and thereby create a new block)." — https://en.bitcoin.it/wiki/Help:FAQ#How_are_new_bitcoins_created.3F

"Mining is the process of spending computation power to secure Bitcoin transactions against reversal and introducing new Bitcoins to the system.

Technically speaking, mining is the calculation of a hash of the a block header, which includes among other things a reference to the previous block, a hash of a set of transactions and a nonce. If the hash value is [correct], a new block is formed and the miner gets the newly generated Bitcoins. [Otherwise], a new nonce is tried, and a new hash is calculated."

The advantage of using such a mechanism consists of the fact, that it is very easy to check a result: Given the payload and a specific nonce, only a single call of the hashing function is needed to verify that the hash has the required properties. Since there is no known way to find these hashes other than brute force, this can be used as a "proof of work" that someone invested a lot of computing power to find the correct nonce for this payload.

This feature is then used in the Bitcoin network to allow the network to come to a consensus on the history of transactions. An attacker that wants to rewrite history will need to do the required proof of work before it will be accepted. And as long as honest miners have more computing power, they can always outpace an attacker. — https://en.bitcoin.it/wiki/Help:FAQ#What_is_mining.3F

Nasz (bardzo uproszczony) blockchain opiera się na mieszance rozwiązań z Bitcoin i Ethereum. W naszym schemacie, transakcja to przekazanie środków między adresami (kwestię podpisywania transakcji, acz bardzo istotną, tutaj ignorujemy). Kwoty reprezentujemy jako liczby naturalne w tysięcznych częściach "monety" (coin).

W praktyce zachętą do włączania transakcji do bloku są opłaty od transakcji, które otrzymuje twórca bloku, tutaj tę kwestię również pomijamy.

```
type Address = Hash
type Amount = Word32
coin :: Amount
coin = 1000  -- Bitcoin dzieli się na 10^8 Satoshi, tu mniej zer

data Transaction = Tx
  { txFrom :: Address
  , txTo :: Address
  , txAmount :: Amount
  } deriving Show

tx1 = Tx
  { txFrom = hash "Alice"
  , txTo = hash "Bob"
  , txAmount = 1*coin
  }
```

Blok składa się z nagłówka i listy transakcji:

```
data Block = Block { blockHdr :: BlockHeader, blockTxs :: [Transaction]}
data BlockHeader = BlockHeader
  {
    parent :: Hash
  , coinbase :: Transaction
  , txroot :: Hash -- root of the Merkle tree
  , nonce :: Hash
  } deriving Show
```

Nagłówek zawiera:

- `parent` — skrót poprzedniego bloku (0 dla pierwszego bloku)
- `txroot` — skrót z korzenia drzewa skrótów transakcji

- `coinbase` — specjalna transakcja zawierająca nagrodę dla twórcy bloku
- `nonce` — dowód pracy, wartość taka, by skrót nagłówka spełniał zadany warunek

Skrótem bloku jest skrót jego nagłówka (skrót drzewa transakcji jest zawarty w nagłówku).

Blok jest poprawnym przedłużeniem łańcucha, którego ostatnim ogniwem jest blok o skrócie `parent`, jeśli jego skrót kończy się (binarnie) liczbą zer wyznaczoną przez parametr `difficulty`, `txroot` jest skrótem korzenia drzewa skrótów utworzonego dla listy transakcji bloku poprzedzonej `coinbase`, zaś `coinbase` spełnia warunki podane poniżej.

```
difficulty = 5
blockReward = 50*coin
validNonce :: BlockHeader -> Bool
validNonce b = (hash b) `mod` (2^difficulty) == 0
```

`coinbase` - specjalna transakcja zawierająca nagrodę dla twórcy bloku

```
coinbaseTx miner = Tx  { txFrom = 0, txTo = miner, txAmount = blockReward }
```

Uzupełnij funkcję

```
mineBlock :: Miner -> Hash -> [Transaction] -> Block
mineBlock miner parent txs = undefined
```

tak aby tworzyła blok stanowiący poprawne przedłużenie łańcucha zakończonego skrótem `parentHash` i zawierający transakcje `txs`.

Stwórz funkcje

```
validChain :: [Block] -> Bool
verifyChain :: [Block] - > Maybe Hash
verifyBlock :: Block -> Hash -> Maybe Hash
```

- `verifyBlock block parentHash` daje w wyniku `Just h`, o ile `block` ma skrót `h` i jest poprawnym następcą bloku o skrócie `parentHash`
- `verifyChain blocks` daje w wyniku `Just h`, o ile na liście `blocks` każdy blok jest poprawnym przedłużeniem nastepujących po nim (argument jest listą bloków od ostatniego do pierwszego), i pierwszy blok na liście ma skrót `h` (dla listy pustej `Just 0`).
- `validChain blocks` daje `True` o ile `blocks` jest poprawnym łańcuchem zgodnie z opisem powyżej.

```
>>> verifyChain [block1, block2]
Nothing
>>> VH <$> verifyChain [block2,block1,block0]
Just 0x0dbea380
```

NB transakcje przechowujemy w naturalnej kolejności, bloki w kolejności odwróconej, tj. najnowszy blok dołączamy na początku listy.

D. Poświadczenia transakcji

Zdefiniuj poświadczenia transakcji, ich generowanie i kontrolę:

```
data TransactionReceipt = TxReceipt
  {  txrBlock :: Hash, txrProof :: MerkleProof Transaction } deriving Show

mineTransactions :: Miner -> Hash -> [Transaction]
                 -> (Block, [TransactionReceipt])

validateReceipt :: TransactionReceipt -> BlockHeader -> Bool
```

```
validateReceipt r hdr = txrBlock r == hash hdr
                        && verifyProof (txroot hdr) (txrProof r)
```

Funkcja `mineTransactions` ma stworzyć poprawny blok zawierający podaną listę transakcji (+coinbase) i listę poświadczeń transakcji, które portem mogą zostać zweryfikowane przez `validateReceipt`.

Przykłady/testy:

```
>>> let charlie = hash "Charlie"
>>> let (block, [receipt]) = mineTransactions charlie (hash block1) [tx1]
>>> block
BlockHeader {
  parent = 797158976,
  coinbase = Tx {
    txFrom = 0,
    txTo = 1392748814,
    txAmount = 50000},
  txroot = 2327748117, nonce = 3}
Tx {txFrom = 2030195168, txTo = 2969638661, txAmount = 1000}
<BLANKLINE>

>>> receipt
TxReceipt {
  txrBlock = 230597504,
  txrProof = MerkleProof (Tx {
    txFrom = 2030195168,
    txTo = 2969638661,
    txAmount = 1000})
  >0xbcc3e45a}
>>> validateReceipt receipt (blockHdr block)
True
```

(niektóre linie zostały złamane dla czytelności, wiążące są testy w `Blockchain-template.hs`)

E. Drukowanie

Uzupełnij dostarczony moduł `PPrint` tak, aby:

```
>>> runShows $ pprListWith pprBlock [block0, block1, block2]
hash: 0x70b432e0
parent: 0000000000
miner: 0x7203d9df
root: 0x5b10bd5d
nonce: 18
Tx# 0x5b10bd5d from: 0000000000 to: 0x7203d9df amount: 50000
hash: 0x2f83ae40
parent: 0x70b432e0
miner: 0x790251e0
root: 0x5ea7a6f0
nonce: 0
Tx# 0x5ea7a6f0 from: 0000000000 to: 0x790251e0 amount: 50000
hash: 0x0dbea380
parent: 0x2f83ae40
miner: 0x5303a90e
root: 0x8abe9e15
```

```
nonce: 3
Tx# 0xbcc3e45a from: 0000000000 to: 0x5303a90e amount: 50000
Tx# 0x085e2467 from: 0x790251e0 to: 0xb1011705 amount: 1000
```

(funkcja `pprBlock` jest zdefiniowana w dostarczonym szkielecie modułu `Blockchain`)

Testy

Komentarze rozpoczynające się od sekwencji `>>>` są wykonywalnymi testami. Można je sprawdzić używając `doctest`:

```
$ doctest HashTree.hs
Examples: 8  Tried: 8  Errors: 0  Failures: 0: 0
$ doctest Blockchain.hs
Examples: 16  Tried: 16  Errors: 0  Failures: 0
```

Wymagania Techniczne

1. Należy oddać pliki `HashTree.hs`, `Blockchain.hs` i `PPrint.hs`
2. Na początku każdego pliku musi się znajdować komentarz z identyfikatorem autora (ze students).
3. Można importować:
   - moduły dostarczone z zadaniem (`Hashable32`, `Utils`)
   - stworzony przez siebie moduł `HashTree`
   - uzupełniony moduł `PPrint`
   - moduły ze standardowego pakietu `base`
4. Do dostarczonych szablonów modułów można dodawać własne funkcje. Nie można natomiast usuwać, ani zmieniać znajdujących się w nich elementów, w szczególności komentarzy zawierających testy. Można zmieniać/usuwać/dodawać argumenty w definicjach funkcji (przy zachowaniu ich typów).

Ocenianie

Zadanie będzie oceniane nie tylko pod kątem poprawności, ale również czytelności kodu i wykorzystania poznanych mechanizmów języka.

Zadanie MUSI być rozwiązane samodzielnie. Wszelkie zapożyczenia muszą być wyraźnie zaznaczone z podaniem źródła.

Zabronione jest oglądanie rozwiązań, jak również wszelkie formy udostępniania własnego rozwiązania innym osobom.

Rozwiązania niesamodzielne będą oceniane na 0p. W wypadku stwierdzenia istotnego podobieństwa dwóch (lub większej liczby) rozwiązań, wszystkie będą oceniane na 0p. Dlatego należy zadbać o utrzymanie prywatności swojego kodu.

Rozwiązania częściowe

Warunkiem uzyskania punktów za rozwiązanie częściowe (nie obejmujące całego zakresu zadania, bądź nie przechodzące wszystkich wymaganych testów) jest wyraźne opisanie zrealizowanego zakresu w komentarzu na początku oddawanego pliku.

Rozwiązaniom częściowym przyznana zostanie znacznie mniejsza liczba punktów niż rozwiązaniom pełnym (zasadniczo **najwyżej** 50%).