Program zaliczeniowy z Haskella (8p)

Dany następujący typ reprezentujący niedeterministyczne automaty skończone nad alfabetem `a` o stanach z `q`

```
data Auto a q = A { states      :: [q]
                  , initStates  :: [q]
                  , isAccepting :: q -> Bool
                  , transition  :: q -> a -> [q]
                  }
```

A. Stwórz moduł `Auto`, eksportujący typ `Auto a q` (ale bez konstruktorów) oraz funkcje

```
accepts :: Eq q => Auto a q -> [a] -> Bool
emptyA :: Auto a ()
epsA :: Auto a ()
symA :: Eq a => a -> Auto a Bool
leftA :: Auto a q -> Auto a (Either q r)
sumA :: Auto a q1 -> Auto a q2 -> Auto a (Either q1 q2)
thenA :: Auto a q1 -> Auto a q2 -> Auto a (Either q1 q2)
fromLists :: (Eq q, Eq a) => [q] -> [q] -> [q] -> [(q,a,[q])] -> Auto a q
toLists :: (Enum a,Bounded a) => Auto a q -> ([q],[q],[q],[(q,a,[q])])
```

takie, że

- `accepts aut w` mówi czy automat `aut` akceptuje słowo `w`;
- `emptyA` rozpoznaje język pusty;
- `epsA` rozpoznaje język złożony ze słowa pustego;
- `symA c` rozpoznaje język `{c}`
- `leftA aut` rozpoznaje ten sam język co `aut`;
- język automatu `sumA aut1 aut2` jest sumą języków dla `aut1` i `aut2`;
- język automatu `thenA aut1 aut2` jest konkatenacją języków dla `aut1` i `aut2`;
- `fromLists` i `toLists` tłumaczą pomiędzy reprezentacją funkcyjną a reprezentacją listową `(stany,startowe,akceptujące,przejścia)`;

tudzież instancję klasy `Show`

```
instance (Show a, Enum a, Bounded a, Show q) => Show (Auto a q) where
```

warto przy tym zadbać aby wypisywana reprezentacja była w miarę możności czytelna, np.

```
*Auto> symA 'x'
fromLists [False,True] [False] [True] [(False,'x',[True])]
```

Funkcja `accepts` powinna działać w czasie liniowym zwn. długość słowa. W każdym razie implementacje wykładnicze będą oceniane niżej.

Wskazówka: przydatne mogą być funkcje

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
any :: (a -> Bool) -> [a] -> Bool
```

Moduł `Auto` nie powinien importować nic poza `Prelude` (które jest importowane domyślnie) i ewentualnie `Data.List`.

Pomocne testy są zawarte w plikach `TestAuto.hs`, `SimpleTests.hs`, można je uruchomić przez `runhaskell` ze swoim modułem `Auto`:

```
Zadanie2017> time runhaskell SimpleTests.hs
null PASS
eps1 PASS
eps2 PASS
chr1 PASS
chr2 PASS
cliq PASS
fork PASS
runhaskell SimpleTests.hs  0,36s user 0,07s system 95% cpu 0,451 total
Zadanie2017>  time runhaskell TestAuto.hs
testSumA
+++ OK, passed 100 tests.
testSumA1
+++ OK, passed 100 tests.
testThenA
+++ OK, passed 100 tests.
runhaskell TestAuto.hs  0,63s user 0,09s system 94% cpu 0,753 total
```

B. Napisz program `RunAuto`, taki, że wywołanie `RunAuto nazwa` wczyta z pliku `nazwa` opis automatu i słowo i odpowie True lub False w zależności czy automat akceptuje słowo (oczywiście program powinien działać dla dowolnej poprawnej nazwy pliku).

W tej części zadania stanami są liczby naturalne, alfabet składa się z liter [A-Z].

Format pliku wejściowego

```
liczba stanów
lista stanów startowych
lista stanów akceptujących
stan symbole stan ... stan
...
stan symbole stan ... stan
słowo
```

na przykład

```
4
[1]
[3,4]
1 C 1 2
1 AB 1
2 B 3
3 A 4
ABABABACBA
```

automat rozpoznaje język słów złożonych z liter `A,B,C`, kończących się `CBA`, zatem program powinien odpowiedzieć `True`.

Puste linie ignorujemy; w przypadku błędnego wejścia program powinien odpowiedzieć `BAD INPUT` (ewentualnie z komunikatem diagnostycznym)

** Wskazówki: **

Do wczytywania może się przydać funkcja `readMaybe` z modułu `Text.Read`

Do wypisywania automatu może się przydać

```haskell
newtype Alpha = Alpha Char deriving (Eq)
instance Bounded Alpha where
    minBound = Alpha 'A'
    maxBound = Alpha 'Z'
instance Enum Alpha where ...
instance Show Alpha where ...
```

Programy będą oceniane nie tylko pod kątem poprawności, ale także czytelności i elegancji.

Należy przesłać dwa pliki: `Auto.hs` i `RunAuto.hs` Na mniejszą liczbę punktów (ca 3) można oddać samą część A.