Assignment 4

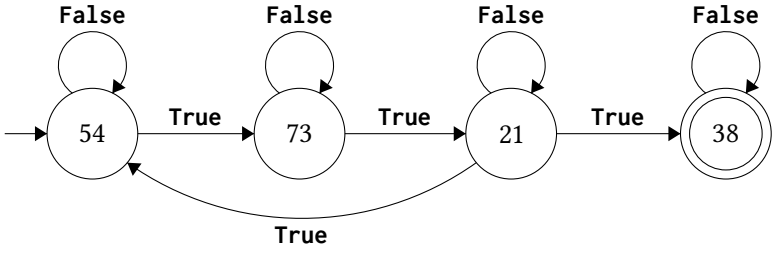
Ethan Poole
LING 185A: Comp. Ling. I
Due: 29 April 2019

Instructions: Download `FiniteState.hs` and `Assignment04.hs` from the course website into the same directory on your computer. The **`import`** line near the top of `Assignment04.hs` imports all of the definitions from `FiniteState.hs`, which you may then use in your assignment. Please submit your assignment as one file: a modified version of `Assignment04.hs`.

1 *5 points*

Please convert the following graphical representation of an FSA into our Haskell format with the type `Automaton`:

(1)



Specifically, define `fsa_1` to be the appropriate thing of type `Automaton` to represent this FSA. You can use the functions `recognize` and `hat` to test the behavior of your implementation:

(2) **Example behavior:**

```
recognize fsa_1 [True, False, True, True, False] ⟹* True
recognize fsa_1 [True, True, True, False, True, True] ⟹* False
hat (transitions fsa_1) 54 [True, False, True, True] ⟹* [38, 54]
hat (transitions fsa_1) 73 [True, False, True, True] ⟹* [73]
```

2 *25 points*

Please construct the following FSAs in Haskell with the type `Automaton`. You may choose to use whatever you want as the state type (though `Int` is a natural choice in most cases). You can use `recognize` to test the behavior of your implementation.

(3) a. Construct an FSA called `fsa_2` that has `SegmentCV` as its alphabet, and generates all and only those sequences that contain at least two Cs.

Example behavior:

```
recognize fsa_2 [C,C,C,C] ⟹* True
recognize fsa_2 [V,V,C,V] ⟹* False
```

b. Construct an FSA called `fsa_3` that has `SegmentCV` as its alphabet, and generates all and only those sequences that have an odd number of Cs and an even number of Vs (treating zero as an even number).

Example behavior:

```
recognize fsa_3 [C] ⟹* True
recognize fsa_3 [C, V, V] ⟹* True
recognize fsa_3 [C, C, V] ⟹* False
recognize fsa_3 [C, C, V, C, V] ⟹* False
```

c. Construct an FSA called `fsa_4` that has `SegmentCV` as its alphabet, and generates all and only those sequences that have C as their third-to-last symbol.

Example behavior:

```
recognize fsa_4 [C, C, C, V, C] ⟹* True
recognize fsa_4 [C, C, V, V, C] ⟹* False
```

d. Construct an FSA called `fsa_5` that has `SegmentPKIU` as its alphabet, which enforces a simple kind of vowel harmony: treating `WB` as the word-boundary symbol, all the vowels within a word must be identical to each other. Any sequences built out of the symbols `P`, `K`, `I`, `U`, and `WB` are allowed

as long as they satisfy this requirement. This includes some strange ones such as those that contain two adjacent "word boundaries", and those including "words" that contain no vowels, etc.; the goal here is just to isolate the vowel-harmony requirement itself.

Example behavior:

`recognize fsa_5 [P, K, I, K, WB, U, P, U]` $\Longrightarrow^*$ **`True`**
`recognize fsa_5 [P, K, I, K, U, P, U]` $\Longrightarrow^*$ **`False`**
`recognize fsa_5 [K, P, P, P]` $\Longrightarrow^*$ **`True`**
`recognize fsa_5 [K, I, P, U]` $\Longrightarrow^*$ **`False`**

e. Construct an FSA called `fsa_6` that has `SegmentPKIU` as its alphabet, and generates all and only those sequences that (i) do not contain `WB`, and (ii) satisfy the requirement that `U` can only appear somewhere after a `P`.

Example behavior:

`recognize fsa_6 [P, U]` $\Longrightarrow^*$ **`True`**
`recognize fsa_6 [U, P]` $\Longrightarrow^*$ **`False`**
`recognize fsa_6 [P, K, K, K, K, K, K, U]` $\Longrightarrow^*$ **`True`**
`recognize fsa_6 [K, K, K, K, K, K, K, K]` $\Longrightarrow^*$ **`True`**