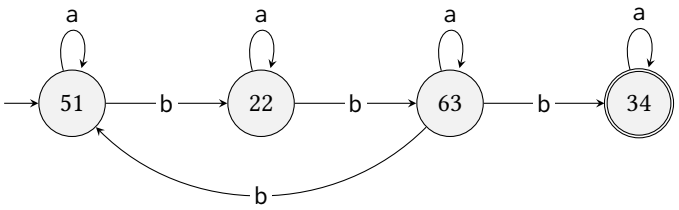
Assignment 6 (Due: Fri Oct 22 at 5pm) Computational Linguistics (LING 455)

1 Programming FSA's

Use this FSA to answer the questions that follow:



(1) Translate this FSA into its Haskell representation:

```haskell
myFSA :: FSA Char -- an FSA whose symbols are Char's
myFSA = (states, syms, i, f, delta)
  where states = undefined
        syms   = undefined
        i      = undefined
        f      = undefined
        delta  = undefined
```

Use `validFSA` to check that your definition of `myFSA` is valid, in that it represents a legal FSA:

```haskell
isItValid :: Bool
isItValid = undefined
```

(2) Let's investigate the sorts of `String`'s `myFSA` generates. `W7.hs` defines an `accepts` function which tests whether a FSA accepts/generates a string. (We'll discuss how this works next week.) That is, `accepts` takes an FSA and a string/list of symbols and returns a `Bool` — `True` if the string is accepted by the FSA, `False` otherwise. For the purposes of this HW, you may take `accepts` to have type `FSA Char -> String -> Bool` (in reality, it has a more general type).

`testSuite` defines a list of `String`'s the above FSA might or might not accept. Use `map` or a list comprehension to test, for each `String` in `testSuite`, if it is `accept`-ed by `myFSA`. Put your answer in `testResults`.

```haskell
testSuite :: [String]
testSuite = [str1, str2, str3, str4, str5]
  where str1 = "bab"
        str2 = "aa"
        str3 = "babba"
        str4 = "bbbabb"
        str5 = "bbbabbb"

testResults :: [Bool]
testResults = undefined
```

Inspecting `testResults` in `ghci` may help you ascertain what pattern the FSA encodes. Another way to see how it works is to start tracing paths through it, writing down the strings you can generate by doing so. Pay

special attention to what has to happen if you take the b-labeled transition back to the initial state. Once you feel you understand how the FSA behaves, characterize the strings it accepts inside a comment:

```
{- Your answer goes here -}
```

Challenge extra credit (2 pts). Define a `Regexp` with the same behavior as `myFSA`. Check your answer by calling (e.g.) `match regexpd "bbbabbb"` and `match regexpd "bbbabb"`. Note: this is **hard**!

```
regexpd :: Regexp
regexpd = undefined
```

2 Inventing FSA's

(3) Now, let's go in the other direction. Define an FSA that generates all and only those strings over the alphabet `['a','b']` with an **even number of `'a'`'s**. I suggest that you first sketch it out on paper and then translate your diagram to Haskell. Hint: you only need two states.

```
evenas :: FSA Char
evenas = (states, syms, i, f, delta)
  where states = undefined
        syms   = undefined
        i      = undefined
        f      = undefined
        delta  = undefined
```

Now do the same for strings with an odd number of `'a'`'s. Hint: this FSA should differ from `evenas` only in which state is designated as final.

```
oddas :: FSA Char
oddas = (states, syms, i, f, delta)
  where states = undefined
        syms   = undefined
        i      = undefined
        f      = undefined
        delta  = undefined
```

Use `accepts` and `all` to construct tests for your FSAs. A suite of test items is defined for you in both cases. Hint: use the types to guide you! `all` has type `(a -> Bool) -> [a] -> Bool`: it takes a function from `a` to `Bool` and a list of `a`'s, and returns `True` if every `a` in the list makes the function `True` (and returns `False` otherwise). Here, the `a` type will be `String`.

```
testEven :: Bool
testEven = undefined
  where suite = ["aa","aba","abbabbbb","","aaaaaabaa"]

testOdd :: Bool
testOdd = undefined
  where suite = ["aaa","ba","abbabbba","a","aaaaaaba"]
```