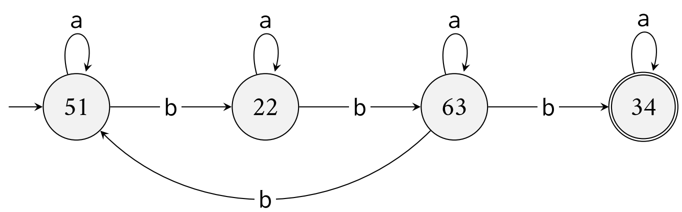
Ling 185A: Assignment 3

Dylan Bumford, Winter 2024

due: Wed, Jan 31

1 Encoding FSAs

(1) Use this FSA to answer the questions that follow:



(a) Translate this FSA into its Haskell representation:

```
myFSA :: FSA
myFSA = FSAWith states syms i f delta
  where states = undefined
        syms   = undefined
        i      = undefined
        f      = undefined
        delta  = undefined
```

(b) Let's investigate the sorts of `String`s `myFSA` generates. `FSA.hs` defines an `accepts` function which tests whether an FSA accepts a string. That is, `accepts` takes an FSA and a string and returns `True` if the string is accepted by the FSA, `False` otherwise.

`testSuite` defines a list of `String`s that the above FSA might or might not accept. Use a list comprehension to pair each `String` in `testSuite` with the `Bool` corresponding to whether or not it is accepted by `myFSA`. Put your answer in `testResults`.

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

testResults :: [(String, 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. Once you feel you understand how the FSA behaves, characterize in your own words the strings it accepts inside this block comment:

```
{-
undefined
-}
```

If you are desperate for things to think about, try to define a `Regexp` with the same behavior as `myFSA`. Check your answer by calling `match myRE "bbbabbb"` and `match myRE "bbbabb"` (you'll need to import the `Regexp` module at the top of this file). Note: this is tricky and completely optional.

```
-- myRE :: Regexp
-- myRE = undefined
```

2 Designing FSAs

(2) 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.

```
evenas :: FSA
evenas = FSAWith 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**.

```
oddas :: FSA
oddas = FSAWith states syms i f delta
  where states = undefined
        syms   = undefined
        i      = undefined
        f      = undefined
        delta  = undefined
```

Use `accepts` (from the FSA module) and `all` (from the `Prelude` imports) to construct tests for your FSAs. A suite of test items is defined for you in both cases. 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"]
```

(3) Vowel harmony (link) is a phonological pattern in which (simplifying greatly) all the vowels within some domain are identical.

Define an FSA that accepts a string if and only if all the vowels within a morpheme are identical. We will work with a very simple alphabet with one consonant `'k'`, two vowels `'i'` and `'u'`, and a space character for morpheme boundaries `'␣'`. Thus, for example, `"ikik␣kukku"`, `"kkuuk"`, and `"␣k␣i␣u"` should all be accepted, but `"kiku"` should not be.

Check your answer using `accepts`.

```haskell
fsaHarmony :: FSA
fsaHarmony = FSAWith states syms i f delta
  where
    states = undefined
    syms   = undefined
    i      = undefined
    f      = undefined
    delta  = undefined
```

```
ghci> accepts fsaHarmony "ikik kukku"
True
ghci> accepts fsaHarmony "kkuuk"
True
ghci> accepts fsaHarmony "  k i u"
True
ghci> accepts fsaHarmony "kiku"
False
```

(4) Define an FSA on the same alphabet accepting exactly the strings in which any `'i'`s appear immediately after a `'k'`. This FSA should accept `"ki"` and `"kiki"`, but not `"kikii"`, or `"ki␣i"`.

Check your answer using `accepts`.

```haskell
fsaKI :: FSA
fsaKI = FSAWith states syms i f delta
  where
    states = undefined
    syms   = undefined
    i      = undefined
    f      = undefined
    delta  = undefined
```

```
ghci> accepts fsaKI "ki"
True
ghci> accepts fsaKI "kiki"
True
ghci> accepts fsaKI "kikii"
False
ghci> accepts fsaKI "ki i"
False
```

(5) Assume our alphabet is contains just `c` and `v`. Write a function `requireCs :: Int -> FSA` that constructs different FSAs for different `Int`s. When given the number `n`, it should return an FSA that accepts all and only those strings that contain exactly `n`-many `'c'`s. Hint: It might be helpful to think about this as two sub-questions: what are the transitions emitting `'c'` that we're allowed to take, and what are the transitions emitting `'v'` that we're allowed to take?

```haskell
requireCs :: Int -> FSA
requireCs n = FSAWith states syms i f (deltaC ++ deltaV)
  where
    states = undefined
    syms   = undefined
    i      = undefined
    f      = undefined
    deltaC = undefined
    deltaV = undefined
```

For instance,

```
ghci> accepts (requireCs 2) "cc"
True
ghci> accepts (requireCs 2) "cv"
False
ghci> accepts (requireCs 2) "cvc"
True
ghci> accepts (requireCs 2) "cvcc"
False
ghci> accepts (requireCs 2) "cvccv"
False
ghci> accepts (requireCs 3) "cvccv"
True
ghci> accepts (requireCs 3) "cvcv"
False
```