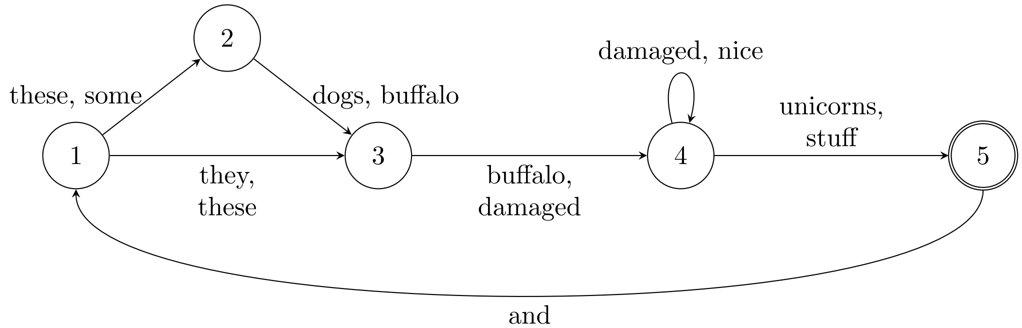
LING185A, Assignment #4

Due date: Wed. 2/8/2016

Download `FiniteState_Stub.hs` from the course website, and rename it to `FiniteState.hs`.[1] This file contains the code for working with finite state grammars that we looked at in class. You will need to submit a modified version of this file. There are `undefined` stubs for each of the questions below.

A. Define `grammar2` to be a new list of grammar rules, analogous to `grammar1`, encoding the finite state machine specified by the diagram below. When two words appear on an arrow, this is just an abbreviation for two distinct arrows, each labeled with a single word. Use the integers `1`, `2`, `3`, `4` and `5` as the "names" of the machine's states.[2]



Use the existing functions to check that `grammar2` is defined correctly. Here are some examples of how things should work:

```
*FiniteState> successors grammar2 1 "these"
[2,3]
*FiniteState> successors grammar2 1 "some"
[2]
*FiniteState> successors grammar2 2 "some"
[]
*FiniteState> takeSteps grammar2 1 ["these","buffalo"]
[3,4]
```

B. Write a function `recognize`, with type `[GrammarRule] -> State -> [String] -> Bool`, such that `recognize g s ws` is `True` iff, in grammar `g`, one can start at state `s` and then make a series of transitions that output the words `ws` and arrive at an ending state. You do not need to write this recursively; just use the existing function `takeSteps`.

```
*FiniteState> recognize grammar1 4 ["the","dog"]
True
*FiniteState> recognize grammar1 1 ["the","dog"]
False
```

[1] Use this name exactly, please. Don't worry, we'll know it's yours.

[2] I'm making use of the handy lexical ambiguity of the word 'buffalo', as made famous by the example sentence 'Buffalo buffalo Buffalo buffalo buffalo bufallo Buffalo bufallo'. See `https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo`.

```
*FiniteState> recognize grammar1 1 ["the","dog","chased","John"]
True
*FiniteState> recognize grammar1 2 ["the","dog","chased","John"]
False
*FiniteState> recognize grammar2 5 ["and","they","damaged","stuff"]
True
*FiniteState> recognize grammar2 3 ["buffalo","damaged","stuff"]
True
*FiniteState> recognize grammar2 2 ["buffalo","damaged","stuff"]
True
*FiniteState> recognize grammar2 2 ["damaged","damaged","stuff"]
False
```

You may find the built-in function `any` helpful. You can use it as if it has type `(a -> Bool) -> [a] -> Bool`, and it will return `True` iff at least one element of the given list satisfies the given predicate. (But of course, you could also write this function yourself, in just a few minutes. It's essentially the `contains` function from Assignment #2.)

C. Write a function `generate`, with type `[GrammarRule] -> Numb -> [StrucDesc]`, analogous to the function of the same name from the last assignment: it should generate all the well-formed structural descriptions up to the given "size", where now we understand the size of a structural description to be the length of the sequence of states it describes. (So the number of words will be one less than this number.) Do not use `wellFormed` here!

```
*FiniteState> map pf (generate grammar1 (S(S(S Z))))
["","left","cat left","dog left","John left","cat","the cat","dog","the dog","John","chased John",
"admired John","left John"]
*FiniteState> map pf (generate grammar2 (S(S(S Z))))
["","unicorns","buffalo unicorns","damaged unicorns","damaged unicorns","nice unicorns","stuff",
"buffalo stuff","damaged stuff","damaged stuff","nice stuff"]
```

You can do this however you like, but it's fairly straightforward to do by adapting the functions `extendByOne` and `extend` from last week as well. One important difference though is that whereas with bigrams it made sense to write a function

```
predecessors :: [GrammarRule] -> String -> [String]
```

here the natural analog is

```
predecessors :: [GrammarRule] -> State -> [(State,String)]
```

which would work like this:

```
*FiniteState> predecessors grammar2 4
[(3,"buffalo"),(3,"damaged"),(4,"damaged"),(4,"nice")]
*FiniteState> predecessors grammar2 3
[(1,"they"),(1,"these"),(2,"dogs"),(2,"buffalo")]
*FiniteState> predecessors grammar2 1
[(5,"and")]
```

This is the first time *ordered pairs* have come up, but the idea should be familiar from mathematics: you can think of the type `(State,String)` as the cartesian product of the type `State` and the type `String` — i.e. something like `State`×`String`. So `[(State,String)]` is the type of lists of ordered pairs, where each pair has a state as its first coordinate and a string as its second coordinate. And if you use a `predecessors` function that works like this, you might find it handy that when are using a lambda to define a function whose argument is a pair, we can write it in the form `(\(x,y) -> ...)`. For example:

```
*FiniteState> map (\(x,y) -> x + y) [(2,3), (4,5), (6,7)]
[5,9,13]
```

D. Write a function `parse`, with type `[GrammarRule] -> State -> [String] -> [StrucDesc]`, such that `parse g s ws` is a list of all the well-formed (according to `g`) structural descriptions that start in state `s` that output the words `ws`.

```
*FiniteState> parse grammar1 4 ["the","dog"]
[NonLast 4 "the" (NonLast 5 "dog" (Last 6))]
*FiniteState> parse grammar1 1 ["the","dog"]
[]
*FiniteState> parse grammar2 1 ["these","buffalo","damaged","unicorns"]
[NonLast 1 "these" (NonLast 2 "buffalo" (NonLast 3 "damaged" (NonLast 4 "unicorns" (Last 5)))),
 NonLast 1 "these" (NonLast 3 "buffalo" (NonLast 4 "damaged" (NonLast 4 "unicorns" (Last 5))))]
*FiniteState> parse grammar2 4 ["stuff"]
[NonLast 4 "stuff" (Last 5)]
*FiniteState> parse grammar2 2 ["dogs"]
[]
*FiniteState> parse grammar2 2 []
[]
*FiniteState> parse grammar2 5 []
[Last 5]
```

This is somewhat tricky. Some hints:

- You need to write this recursively on the `[String]` argument, following a similar pattern to `takeSteps`. That is, you need to answer these two questions:
  - When the `[String]` argument is `[]`, what is the desired result?
  - When the `[String]` argument is `w:ws`, how do we build the desired result of out the result(s) of some recursive call(s) to `parse` to which we pass `ws` as the `[String]` argument?

  Think about the way these questions get answered in `takeSteps`, and ask yourself how to adapt them.
- Notice that the states that appear at the *ends* of the structural descriptions in the list `parse g s ws` are the states in the list `takeSteps g s ws`.
- In a situation where word `w` can get you from state `s1` to state `s2` in grammar `g`, the elements of the list `takeSteps g ws s2` *are* elements of the list `takeSteps g (w:ws) s1`. The elements of the list `parse g ws s2`, however, are not themselves elements of the list `parse g (w:ws) s1` — although they are a useful starting point.

Notice that in `grammar2` there are *four* structural descriptions starting at state `1` for the word-sequence 'these buffalo damaged stuff and these buffalo damaged stuff'.

Things to think about

- Recall that in a bigram grammar, knowing that a certain word-sequence is generated doesn't leave any questions open about *how* it is generated, and accordingly there's no such thing as "being generated in two distinct ways". But with a finite-state grammar we can ask not just "Is this word-sequence generated?" (as `recognize` does) but also "How is this word-sequence generated?" (as `parse` does). **A structural description is an answer to this "how" question.** Having multiple answers to this "how" question (i.e. having `parse` return multiple structural descriptions) gives us at least the beginnings of a system that has a hope of accounting for semantic ambiguities, i.e. one word-sequence being paired with two meanings.
- At the end of the last assignment we saw how it was fairly easy to imagine how a learner could, in principle, construct a bigram grammar on the basis of some finite amount of input, in such a way that this constructed grammar generated things that were not in the input. Notice that if we were to try to do the same thing with finite state grammars, however, it's much more difficult to see which particular grammar we should contruct from any given input — the states are not visible in the input, so it's

hard to know how to even get started deciding on what the states of a constructed grammar should be. (So imagine how difficult things get when the hidden structure gets even more complex . . .)