LING185A, Assignment #4

Due date: Wed. 2/7/2018

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.

One tiny change from what you saw in class: the function we wrote called `takeSteps` has been renamed `takeStepsBack` (for reasons that either are immediately obvious or will be very shortly). It will probably be useful to make sure you understand how that works first. The crucial idea underlying the recursive line is this: the things you can reach by taking steps forward from state `s` and emitting `w:ws`, are the things you can reach by taking steps forward from any `nextState` in the list `successors g s w` and emitting `ws`.

Also, have a look at the `predecessors` function, which we did not get around to discussing in class.

In some of the exercises below you will find the `elem` function useful. Recall that this can be used to check whether a certain element appears in a list.

```
Prelude> elem 3 [2,3,4,5]
True
Prelude> elem 6 [2,3,4,5]
False
Prelude> elem "three" ["two","three","four","five"]
True
Prelude> elem "hello" ["two","three","four","five"]
False
Prelude> elem True [False,False,True,False]
True
Prelude> elem True [False,False,False,False]
False
```

1 Recognizing

A. Write a function `takeStepsBack :: [GrammarRule] -> [String] -> [State]`, which produces a list of all the states in the given grammar that you can start at, then emit the given sequence of word, and end up at a valid ending state. This functions deals with "ending parts" of a potentially generated word-sequence, in the way that `takeStepsForward` deals with "starting parts". The difference in the types of these two functions is due to the fact that a `[GrammarRule]` already specifies valid ending states, but does not specify starting states. Like `takeStepsForward`, it's OK if this function sometimes produces duplicates if there are multiple relevant "routes" through the grammar. (The crucial idea underlying the recursive step here is this: the states from which you can get to an end state by emitting `w:ws`, are the states in the list `predecessors g w nextState` for any `nextState` from which you can get to an end state by emitting `ws`.)

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

```
*FiniteState> takeStepsBack grammar3 ["damaged","stuff"]
[3,4]
*FiniteState> takeStepsBack grammar3 ["buffalo","damaged","stuff"]
[2,3]
*FiniteState> takeStepsBack grammar3 ["these","buffalo","damaged","stuff"]
[1,1]
*FiniteState> takeStepsBack grammar4 ["a"]
[3]
*FiniteState> takeStepsBack grammar4 ["b"]
[2,3]
*FiniteState> takeStepsBack grammar4 ["b","a"]
[2,3]
*FiniteState> takeStepsBack grammar4 ["b","b","a"]
[1,2,3]
```

B. Using `takeStepsBack`, write a function `recognizeBackward` with type `[GrammarRule] -> State -> [String] -> Bool`, such that `recognizeBackward 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. (In other words, the result should be `True` iff there is a structural description `sd` such that (i) `wellFormed g sd` is `True`, (ii) `firstState sd` is `s`, and (iii) `pf sd` is `ws`.) You do not need to write this recursively.

```
*FiniteState> recognizeBackward grammar1 4 ["the","dog"]
True
*FiniteState> recognizeBackward grammar1 1 ["the","dog"]
False
*FiniteState> recognizeBackward grammar1 1 ["the","dog","chased","John"]
True
*FiniteState> recognizeBackward grammar1 2 ["the","dog","chased","John"]
False
*FiniteState> recognizeBackward grammar3 5 ["and","they","damaged","stuff"]
True
*FiniteState> recognizeBackward grammar3 3 ["buffalo","damaged","stuff"]
True
*FiniteState> recognizeBackward grammar3 2 ["buffalo","damaged","stuff"]
True
*FiniteState> recognizeBackward grammar3 2 ["damaged","damaged","stuff"]
False
*FiniteState> recognizeBackward grammar3 3 ["damaged","damaged","stuff"]
True
*FiniteState> recognizeBackward grammar3 4 ["damaged","damaged","stuff"]
True
```

C. Now using the existing function `takeStepsForward`, write a function `recognizeForward` which does exactly the same thing as `recognizeBackward` does. The only difference is that this time you should use `takeStepsForward` rather than `takeStepsBack`. Notice that whereas `takeStepsBack` "knows about" both `Step` rules and `End` rules, `takeStepsForward` only knows about `Step` rules.

2 Parsing

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 the `recognize` functions do) but also "How is this word-sequence generated?". A structural description is an answer to this "how" question, and *parsing* is the task of finding structural descriptions on the basis of the word-sequence that they underlie. Having multiple answers to this "how" question (i.e. having the `parse` function below return multiple structural descriptions) gives us at least the beginnings of a system that accounts for semantic ambiguities, i.e. one word-sequence being paired with two meanings.

D. Write a function `parse`, with type `[GrammarRule] -> [String] -> [StrucDesc]`, such that `parse g ws` is a list of all the structural descriptions `sd` such that (i) `wellFormed g sd` is `True`, and (ii) `pf sd` is `ws`. Note that there is no relevant notion of "starting state" here. The way to do this is (obviously?) to follow the pattern in `takeStepsBack`: the only difference is that at the core of `takeStepsBack` there's a function of type `State -> [State]` which is mapped over a list, whereas here you need to use an analogous function of type `StrucDesc -> [StrucDesc]`.

```
*FiniteState> parse grammar1 ["the","dog"]
[NonLast 4 "the" (NonLast 5 "dog" (Last 6))]
*FiniteState> parse grammar3 ["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 grammar3 ["damaged","unicorns"]
[NonLast 3 "damaged" (NonLast 4 "unicorns" (Last 5)),
 NonLast 4 "damaged" (NonLast 4 "unicorns" (Last 5))]
*FiniteState> parse grammar3 []
[Last 5]
*FiniteState> parse grammar4 ["a","b","b"]
[NonLast 1 "a" (NonLast 1 "b" (NonLast 2 "b" (Last 3))),
 NonLast 3 "a" (NonLast 3 "b" (NonLast 3 "b" (Last 3)))]
*FiniteState> parse grammar4 ["a","b"]
[NonLast 3 "a" (NonLast 3 "b" (Last 3))]
*FiniteState> parse grammar4 ["a","b","b"]
[NonLast 1 "a" (NonLast 1 "b" (NonLast 2 "b" (Last 3))),
 NonLast 3 "a" (NonLast 3 "b" (NonLast 3 "b" (Last 3)))]
*FiniteState> parse grammar4 ["b","a","b"]
[NonLast 2 "b" (NonLast 3 "a" (NonLast 3 "b" (Last 3))),
 NonLast 3 "b" (NonLast 3 "a" (NonLast 3 "b" (Last 3)))]
*FiniteState> parse grammar4 ["b","b","b"]
[NonLast 1 "b" (NonLast 1 "b" (NonLast 2 "b" (Last 3))),
 NonLast 1 "b" (NonLast 2 "b" (NonLast 3 "b" (Last 3))),
 NonLast 2 "b" (NonLast 3 "b" (NonLast 3 "b" (Last 3))),
 NonLast 3 "b" (NonLast 3 "b" (NonLast 3 "b" (Last 3)))]
```

3 Generating

E. Write a function `generate`, with type `[GrammarRule] -> Numb -> [StrucDesc]`, analogous to the function of the same name from last week: 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.) Again, note that there is no relevant notion of "starting state" here. Do not use `wellFormed`!

```
*FiniteState> map pf (generate grammar1 (S (S (S Z))))
[[],["left"],["cat"],["dog"],["John"],["cat","left"],["dog","left"],["John","left"],["the","cat"],
 ["the","dog"],["chased","John"],["admired","John"],["left","John"]]
*FiniteState> map pf (generate grammar2 (S (S (S (S (S Z))))))
[[],["runs"],["run"],["often","runs"],["often","run"],["student","often","runs"],
 ["students","often","run"],["the","student","often","runs"],["the","students","often","run"]]
*FiniteState> map pf (generate grammar3 (S (S (S Z))))
[[],["unicorns"],["stuff"],["buffalo","unicorns"],["damaged","unicorns"],["damaged","unicorns"],
 ["nice","unicorns"],["buffalo","stuff"],["damaged","stuff"],["damaged","stuff"],["nice","stuff"]]
*FiniteState> map pf (generate grammar4 (S (S (S Z))))
[[],["b"],["a"],["b"],["b","b"],["b","a"],["a","a"],["b","a"],["b","b"],["a","b"],["b","b"]]
*FiniteState> length (generate grammar4 (S (S (S (S (S Z))))))
63
```

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. An important difference though is that this week's `predecessors` function can't be used here in the same way that the function of this name could be used last week (because of the way structural descriptions now have more information in them than a list of symbols/words). Instead, you might decide to write a function `incomingSteps`, with type `[GrammarRule] -> State -> [(State,String)]`, which works like this:

```
*FiniteState> incomingSteps grammar3 4
[(3,"buffalo"),(3,"damaged"),(4,"damaged"),(4,"nice")]
*FiniteState> incomingSteps grammar3 3
[(1,"they"),(1,"these"),(2,"dogs"),(2,"buffalo")]
*FiniteState> incomingSteps grammar3 2
[(1,"these"),(1,"some")]
*FiniteState> incomingSteps grammar3 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]
```