LING185A, Assignment #3

Due date: Wed. 2/1/2017

Download `Bigrams_Stub.hs`, and rename it to `Bigrams.hs`. (Please use this name exactly.[1]) This file contains the definitions of the `GrammarRule` and `StrucDesc` types for the very simple bigram grammars that we talked about in class. You will need to submit a modified version of this file.

The file you have downloaded contains "stubs" for all of the functions that you will need to write for this assignment: for this I have used `undefined`, which is sort of a magic word that means "please, GHCi, just pretend there is some expression here that has the type you expect to be here, and if you promise not to bother me about that then I promise not to ask to you actually evaluate that expression".

Of course, the fact that I have provided a stub that looks like

```
f = undefined
```

does not mean that you need to leave the `f =` part as it is and use explicit lambdas on the right hand side; you are, naturally, free to change it to

```
f x y = ...
```

or whatever. Just make sure that you define a function with the specified name and type.

Background

Some list functions to know about

- The binary operator `++` can be used to concatenate two lists. Its type is `[a] -> [a] -> [a]`. Its analogous to the `append` function that you wrote last week for `NumbLists`. And because strings are just lists of characters — specifically, `String` is a synonym for the type `[Char]` — you can use this operator to concatenate strings.

```
Prelude> [3,4,5] ++ [7,8,9]
[3,4,5,7,8,9]
Prelude> ["one","two","three"] ++ ["hello","world"]
["one","two","three","hello","world"]
Prelude> "one" ++ "two"
"onetwo"
```

- The function `words :: String -> [String]` splits a string up into words, taking spaces to be word boundaries.

```
Prelude> words "here are some words"
["here","are","some","words"]
Prelude> words "here are some more"
["here","are","some","more"]
```

[1] Yes, exactly. No, don't worry, we will know it's yours, I promise.

```
Prelude> words "i love recursion"
["i","love","recursion"]
Prelude> words "recursion"
["recursion"]
```

- The function `applyToAll :: (a -> b) -> [a] -> [b]` that we (hastily) wrote in class is included at the top of `Bigrams_Stub.hs`. But it's equivalent to the built-in function `map`. You're free to use either for this assignment, but in future I will probably just use `map`.

```
Prelude> map (\x -> x+2) [10,11,12]
[12,13,14]
Prelude> map words ["i love recursion", "colorless green ideas"]
[["i","love","recursion"],["colorless","green","ideas"]]
Prelude> map even [2,3,4,5,6,7]
[True,False,True,False,True,False]
```

- The function `concat` can be used to "flatten" a list of lists into a single list. You can think of it as having type `[[a]] -> [a]`. It recursively combines lists two at a time using `++` (and the empty list), just like the way your `total` function from last week recursively combines numbers two at a time using `add` (and `Z`).

```
Prelude> concat [[3,4,5],[7,8,9],[10,20,30]]
[3,4,5,7,8,9,10,20,30]
Prelude> concat ["aaa","bb","c"]
"aaabbc"
```

Some other helper functions

I've written a couple of helper functions which make it (slightly) easier (perhaps) to work with these grammars, and to provide some more examples of how we write recursive functions on lists. These are the functions `successors`, which returns a list of all the words that a grammar allows to follow a given word, and `enders`, which returns a list of all the words that a grammar allows a sentence to end with.

```
*Bigrams> successors grammar1 "hamsters"
["run","walk"]
*Bigrams> successors grammar1 "slowly"
[]
*Bigrams> successors grammar2 "very"
["small","very"]
*Bigrams> enders grammar1
["run","quickly","slowly"]
```

I've also added a modified version of the `wellFormed` function that we wrote in class which makes use of these helper functions. The logic is entirely unchanged though. (And the version from class is still there, commented out.)

1 Working with structural descriptions

Notice that for the questions in this section, we don't care at all about grammars — we're just dealing with structural descriptions in their own right. Some of them will be well-formed with respect to certain grammars, and others will not be. A structural description that is not well-formed with respect to a particular

grammar, or even that is not well-formed with respect to any grammar that we have ever written down, is still a structural description.

A. Write a function `sdLength`, with type `StrucDesc -> Numb` which computes the number of words in a structural description.

```
*Bigrams> sdLength sd1
S (S (S (S Z)))
*Bigrams> sdLength sd2
S (S Z)
*Bigrams> sdLength sd3
S (S Z)
```

B. Write a function `lastWord :: StrucDesc -> String` which returns the last word in a structural description.

```
*Bigrams> lastWord sd1
"run"
*Bigrams> lastWord sd2
"run"
*Bigrams> lastWord sd3
"hamsters"
*Bigrams> applyToAll lastWord [sd1,sd2,sd3]
["run","run","hamsters"]
```

C. Write a function `pf :: StrucDesc -> String` which concatenates the words of a given structural description in the order they appear, with spaces in between. Remember that since strings are just lists, you can concatenate them with `++`.

```
*Bigrams> pf sd1
"the small hamsters run"
*Bigrams> pf sd2
"chipmunks run"
*Bigrams> pf sd3
"run hamsters"
*Bigrams> pf (Last "abc")
"abc"
*Bigrams> applyToAll pf [sd1,sd2,sd3]
["the small hamsters run","chipmunks run","run hamsters"]
```

D. Write a function `prependWord :: StrucDesc -> String -> StrucDesc` which adds the given string to the front of a structural description. (Hint: Notice that the result will always contain more than one word. This makes it *very* simple!)

```
*Bigrams> prependWord sd2 "hello"
NonLast "hello" (NonLast "chipmunks" (Last "run"))
*Bigrams> prependWord (Last "world") "hello"
NonLast "hello" (Last "world")
```

2 Working with grammars

E. Write a function `predecessors :: [GrammarRule] -> String -> [String]` which is analogous to `successors`, but returns a list of all the words that are allowed to come *before* the given word. If you

copy the code for `successors`, you won't have to make many changes.

```
*Bigrams> predecessors grammar1 "quickly"
["run","walk"]
*Bigrams> predecessors grammar1 "hamsters"
["the","small"]
*Bigrams> predecessors grammar1 "hello"
[]
*Bigrams> predecessors grammar1 "very"
["the"]
*Bigrams> predecessors grammar2 "very"
["the","very"]
```

F. Write a function `rulesFromSentence :: [String] -> [GrammarRule]` which produces a list of grammatical rules on the basis of one sentence of input. The sentence is encoded as the given list of strings, where each string is one word. The resulting grammar should simply allow all and only the bigrams that were "observed" in the sentence, and should allow as an endpoint just the one word that was observed in the sentence-final position. It doesn't matter if the resulting grammar contains duplicate rules. Although it's a bit arbitrary, let's say that if this function is given a list that contains no strings, then the result should be a list that contains no grammar rules.

```
*Bigrams> rulesFromSentence ["big", "bad", "wolf"]
[Step "big" "bad",Step "bad" "wolf",End "wolf"]
*Bigrams> rulesFromSentence (words "a b c c d")
[Step "a" "b",Step "b" "c",Step "c" "c",Step "c" "d",End "d"]
```

G. Write a function `rulesFromText :: [String] -> [GrammarRule]` which produces a list of grammatical rules on the basis of a *list of sentences* provided as input. Each string in the given list of strings is one *sentence*, whose words are separated by spaces. What this function should do is simply use `rulesFromSentence` to produce a list of rules from each sentence, and then throw all the resulting rules together. Again, it's OK if there are duplicates. (Hint: You do not need to do this using recursion; all you need to do is stitch things together using `words`, `applyToAll`/`map` and `concat`.)

```
*Bigrams> rulesFromText ["a b c c b d", "a c"]
[Step "a" "b",Step "b" "c",Step "c" "c",Step "c" "b",
 Step "b" "d",End "d",Step "a" "c",End "c"]
```

3 Putting it all together: Generating grammatical structural descriptions

For the questions in this section, you can "forget about" the fact that a grammar takes the form of a list of rules. You can simplify your thinking by just using a name like `g` for arguments of type `[GrammarRule]` and simply knowing that it's the kind of thing that can be passed to functions like `predecessors` and `enders`.

H. Write a function `extendByOne :: [GrammarRule] -> StrucDesc -> [StrucDesc]` which produces all structural descriptions that (a) contain the given structural description as an *immediate subpart* (i.e. every element of `extendByOne g sd` is of the form `NonLast s sd` for some string `s`), and (b) are valid according to the given grammar if the given structural description is. In other words, this function should extend the given structural description only in ways that are allowed by the grammar; but if it already contains invalid transitions or an invalid ending point, then that's not our business to worry about here. (There's a nearly-complete stub version of this provided, which we wrote in class. You just need to replace `undefined` with an appropriate expression of type `String -> StrucDesc`.)

```
*Bigrams> extendByOne grammar1 sd4
[NonLast "run" (Last "quickly"),NonLast "walk" (Last "quickly")]
*Bigrams> map pf (extendByOne grammar1 sd4)
["run quickly","walk quickly"]
*Bigrams> map pf (extendByOne grammar1 (NonLast "hamsters" (Last "run")))
["the hamsters run","small hamsters run"]
```

I. Write a function `extend :: [GrammarRule] -> Numb -> StrucDesc -> [StrucDesc]` such that `extend g n sd` produces all the structural descriptions that can be produced from `sd` by at most `n` (but perhaps zero) steps of `extendByOne`. Since the things that can be produced by zero steps are just the argument `sd` itself, this should *always* be in the result list. (Hint: You should write this via recursion on the `Numb` argument. The things that can be reached from `sd` in between zero and `S n` steps are (a) `sd` itself, along with (b) all the things that can be reached from a one-step extension of `sd` in between zero and `n` steps.)

```
*Bigrams> extend grammar1 (S Z) sd4
[Last "quickly",NonLast "run" (Last "quickly"),NonLast "walk" (Last "quickly")]
*Bigrams> extend grammar1 Z sd4
[Last "quickly"]
*Bigrams> map pf (extend grammar1 (S Z) sd4)
["quickly","run quickly","walk quickly"]
*Bigrams> map pf (extend grammar1 (S (S Z)) sd4)
["quickly","run quickly","hamsters run quickly","walk quickly","hamsters walk quickly"]
```

J. Write a function `generate :: [GrammarRule] -> Numb -> [StrucDesc]` which produces all the structural descriptions that are well-formed according to the given grammar and contain at most the given number of words. You should not need to write this recursively; all the relevant recursive work has already been done in `extend`. Use `enders` to get started. Do *not* use `wellFormed` here! (NB: `length` is a built-in function that computes the length of a list, in "normal" numbers. I'm using it below here just to provide something that you can try in order to check that your function is producing the desired results, i.e. you should get results that match the lengths below. You do not need to use `length` in your code.)

```
*Bigrams> map pf (generate grammar1 (S Z))
["run","quickly","slowly"]
*Bigrams> map pf (generate grammar1 (S (S Z)))
["run","hamsters run","quickly","run quickly","walk quickly","slowly","walk slowly"]
*Bigrams> length (generate grammar1 (S (S (S (S Z)))))
20
*Bigrams> length (generate grammar1 (S (S (S (S (S Z))))))
27
*Bigrams> length (generate grammar2 (S (S (S (S (S Z))))))
28
*Bigrams> length (generate grammar2 (S (S (S (S (S (S Z)))))))
36
*Bigrams> generate grammar1 (S Z)
[Last "run",Last "quickly",Last "slowly"]
*Bigrams> generate grammar1 Z
[]
```

Things to think about ...

Notice that, having written both `generate` and `rulesFromText`, we've put together a very simple system that can "go beyond its input" — in *something like* the way that human children produce sentences that they have never heard before, very broadly speaking.

```
*Bigrams> map pf (generate (rulesFromText ["a b c c b d", "a c"]) (S (S (S Z))))
["d","b d","a b d","c b d","c","b c","a b c","c b c","c c","b c c","c c c","a c c","a c"]
```

And notice that this was very easy to do, precisely because the assumptions we're making (for now!) about grammars are exceedingly simple.