# LING185A, Assignment #3

Due date: Wed. 1/31/2018

Download `Bigrams_Stub.hs`, and rename it to `Bigrams.hs`. (Please use this name exactly.[1]) You will need to submit a modified version of this file.

This file contains the definitions of the `GrammarRule` and `StrucDesc` types for our very simple bigram grammars, and the functions that we wrote in class for working with them (`wellFormed`, `pf`, etc.). I've added some comments describing what's going on. Check over what's there to make sure you understand it.

The file you have downloaded also 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. It's analogous to the `append` function that you wrote last week for `NumbLists`. We've seen it used to concatenate strings before, which was actually a special case: a string is just a list of characters. (Specifically, `String` is a synonym for the type `[Char]`.)

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

- 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"
```

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

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

- The function `applyToAll :: (a -> b) -> [a] -> [b]` that we 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"
```

- A common pattern that you will find a use for below involves combining `concat` and `map` in a certain way. Suppose you have a function `f` with a type of the form `a -> [b]`, i.e. it takes an `a` and produces a list of `b`s, and you also have a list `mylist` with type `[a]`, i.e. a list of `a`s; and you want to find all the `b`s that can be produced by `f` from any of the elements of `mylist`. Simply doing `map f mylist` will get you something of type `[[b]]`, but applying `concat` to that result will get you to type `[b]`, as desired. Here are some illustrative examples. Notice that since both `a` and `b` are `String` here, the pattern can be repeated, as shown in the last line.

```
Prelude> let f s = ["x" ++ s, "y" ++ s] in f "a"
["xa","ya"]
Prelude> let f s = ["x" ++ s, "y" ++ s] in f "b"
["xb","yb"]
Prelude> let f s = ["x" ++ s, "y" ++ s] in map f ["a","b"]
[["xa","ya"],["xb","yb"]]
Prelude> let f s = ["x" ++ s, "y" ++ s] in concat (map f ["a","b"])
["xa","ya","xb","yb"]
Prelude> let f s = ["x" ++ s, "y" ++ s] in concat (map f (concat (map f ["a","b"])))
["xxa","yxa","xya","yya","xxb","yxb","xyb","yyb"]
```

# 1 Working with structural descriptions

Note 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.

These should be pretty easy warmups, just a bit more practice. You're not expected to use these functions for anything "useful" later on.

**A.** Write a function `sdMap :: (String -> String) -> StrucDesc -> StrucDesc` which applies the given function to each word in the given structural description. In other words, it should do for structural descriptions what `applyToAll`/`map` does for lists.

```
*Bigrams> sdMap (\s -> s ++ s) sd1
NonLast "thethe" (NonLast "smallsmall" (NonLast "hamstershamsters" (Last "runrun")))
*Bigrams> sdMap (\s -> s ++ "!") sd3
NonLast "run!" (Last "hamsters!")
```

**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> lastWord sd4
"quickly"
*Bigrams> applyToAll lastWord [sd1,sd2,sd3,sd4]
["run","run","hamsters","quickly"]
```

## 2 Working with grammars

**C.** 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"]
```

**D.** 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 ["a","b","c","c","b","d"]
[Step "a" "b",Step "b" "c",Step "c" "c",Step "c" "b",Step "b" "d",End "d"]
```

**E.** Write a function `rulesFromText :: [[String]] -> [GrammarRule]` which produces a list of grammatical rules on the basis of a *list of sentences* provided as input. All this function needs to do is use `rulesFromSentence` to produce a list of rules from each sentence, and then throw all the resulting rules together; you do not need to use any recursion here. Again, it's OK if there are duplicates.

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

# 3 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 `successors`, `predecessors` and `enders`.

**F.** 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. Order in the output list does not matter.

```
*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"]]
```

**G.** Write a function `extend :: [GrammarRule] -> Numb -> StrucDesc -> [StrucDesc]` such that the result of `extend g n sd` is the list containing all 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. Order in the output list does not matter. (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.[2])

```
*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"],["walk","quickly"],
 ["hamsters","run","quickly"],["hamsters","walk","quickly"]]
```

[2]Or, equivalently: (a) `sd` itself, along with (b) all one-step extensions of the things that can be reached from `sd` in between zero and `n'` steps.

**H.** 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 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> length (generate grammar2 (S (S (S (S (S (S (S Z))))))))
44
*Bigrams> generate grammar1 (S Z)
[Last "run",Last "quickly",Last "slowly"]
*Bigrams> generate grammar1 Z
[]
```

## Some 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. Think about what's going on if you try the following, for example:

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

  And notice that this was very easy to do, precisely because the assumptions we're making about grammars are (for now!) exceedingly simple: if the format of your grammatical rules is very simple (and "surfacey"), then it's easy to work out which rules an observed sentence is "telling you" to add to your grammar.

- A fun challenge: write a function with type `[GrammarRule] -> Bool` that checks whether a given grammar contains "loops", i.e. whether it generates infinitely many well-formed structural descriptions. (But notice that a system might still "go beyond its input", in the sense mentioned above, even if it does not have this infinite generative capacity.)

- It may seem strange that the way `generate` works is by constructing a sentence "backwards", extending structural descriptions by adding words at the front. Although it's definitely not impossible, the alternative where we extend structural descriptions by adding words at the end is trickier. Think about why.

- Recall that these grammars do not impose any restrictions on what can appear as the *first* symbol of a structural description, only what can appear at the end. Suppose we added a way for a grammar to

specify what was allowed as the first symbol, say by adding a third option to the `GrammarRule` type like this:

```
data GrammarRule = Step String String | End String | Start String
```

This would make writing the `wellFormed` function slightly annoying: although we could certainly write code to get the job done, `wellFormed` itself would not be straightforwardly recursive in the way that it currently is. Think about why.