LING185A, Assignment #5

Due date: Wed. 2/22/2016

Download `ContextFree_Stub.hs` from the course website, and rename it to `ContextFree.hs`.[1] This file contains the code for working with context free 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. Write a function `brackets`, with type `StrucDesc -> String`, which produces a result like that of `pf` but with brackets added to indicate constituency. There should be one opening bracket and one closing bracket for each `Unary` constituent, and one opening bracket and one closing bracket for each `Binary` constituent (and none for `Leaf` constituents).

```
*ContextFree> brackets sd1
"[John [left Mary]]"
*ContextFree> brackets sd2
"[John [[left] [with [a cat]]]]"
*ContextFree> brackets sd3
"[[the dog] [thinks [John [left Mary]]]]"
*ContextFree> brackets sd4
"[[the dog] [thinks [John [[left] [with [a cat]]]]]]"
```

B. A minor variation: Write a function `labeledBrackets`, with type `StrucDesc -> String`, which produces labeled-bracket representations of the given tree structure where each constituent (including leaves now) has its category specified on its opening bracket. You will need to use the function `show` with type `Category -> String` to get a string representation of a category; see the examples below. (For extra fun, paste your output into the tree-generator here: `http://yohasebe.com/rsyntaxtree/`. Setting "Leaf style" to "None" gives the most appropriate picture for the way we're thinking of things.)

```
*ContextFree> show NP
"NP"
*ContextFree> show S
"S"
*ContextFree> labeledBrackets sd1
"[S [NP John] [VP [V left] [NP Mary]]]"
*ContextFree> labeledBrackets sd2
"[S [NP John] [VP [VP [V left]] [PP [P with] [NP [D a] [N cat]]]]]"
*ContextFree> labeledBrackets sd3
"[S [NP [D the] [N dog]] [VP [V thinks] [S [NP John] [VP [V left] [NP Mary]]]]]"
*ContextFree> labeledBrackets sd4
```

C. Write a function `leftmostWord`, with type `StrucDesc -> String`, which finds the word at the leftmost leaf node of the given tree. (Do not use `pf` here!)

```
*ContextFree> leftmostWord sd1
"John"
```

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

```
*ContextFree> leftmostWord (getSubtree sd1 [1])
"left"
*ContextFree> leftmostWord sd2
"John"
*ContextFree> leftmostWord sd3
"the"
```

D. Write a function `numNPs`, with type `StrucDesc -> Int`, which computes the number of node with category `NP` in the given structural description. (Equality is defined on the type `Cat`, although there are ways to do it without making use of this.)

```
*ContextFree> numNPs sd1
2
*ContextFree> numNPs sd2
2
*ContextFree> numNPs sd3
3
*ContextFree> numNPs sd4
3
*ContextFree> numNPs (Binary VP sd1 sd2)
4
*ContextFree> numNPs (Binary NP sd1 sd2)
5
```

E. Write a function `numViolations`, with type `[GrammarRule] -> StrucDesc -> Int` which computes the number of times a structural description violates the rules of the given grammar. A single violation occurs when a node of the tree neither (a) has two daughter trees in accord with some `BinaryStep` rule, nor (b) has one daughter tree in accord with some `UnaryStep` rule, nor (c) has zero daughter trees in accord with some `End` rule. The result should be `0` exactly when the result of `wellFormed` is `True`.

```
*ContextFree> numViolations grammar1 sd1
0
*ContextFree> numViolations [] sd1
5
*ContextFree> numViolations [] sd3
11
*ContextFree> numViolations grammar1 (Binary S (Leaf NP "Bill") (Leaf NP "cat"))
3
*ContextFree> numViolations grammar1 (Binary S (Leaf NP "Bill") (Leaf VP "cat"))
2
*ContextFree> numViolations grammar1 (Binary S (Leaf NP "John") (Leaf VP "cat"))
1
```

F. Write a function `sdMap :: (String -> String) -> StrucDesc -> StrucDesc` which applies the given function to all of the strings at the given tree's leaf nodes, and returns the new version of the tree modified accordingly. (In other words, this function should do for trees what the standard `map` function does for lists.)

```
*ContextFree> sdMap (\s -> s ++ s) sd1
Binary S (Leaf NP "JohnJohn") (Binary VP (Leaf V "leftleft") (Leaf NP "MaryMary"))
*ContextFree> brackets (sdMap (\s -> s ++ "!") sd2)
"[John! [[left!] [with! [a! cat!]]]]"
```

G. Write a function `longestPath :: StrucDesc -> [Cat]` which computes the sequence of categories

that appear along the longest root-to-leaf path in the given tree. If there are multiple paths that are tied in length, prefer those that branch to the left: in other words, the result in such a situation should be the path whose leaf is further left than the leaf of any other equally long path. There's a built-in `length` function which you can use with type `[a] -> Int`.

```
*ContextFree> longestPath sd1
[S,VP,V]
*ContextFree> longestPath sd2
[S,VP,PP,NP,D]
*ContextFree> longestPath sd3
[S,VP,S,VP,V]
*ContextFree> longestPath sd4
[S,VP,S,VP,PP,NP,D]
*ContextFree> longestPath (Leaf NP "foo")
[NP]
*ContextFree> longestPath (Binary S (Leaf NP "foo") (Leaf VP "bar"))
[S,NP]
```

H. Write a function `allPaths :: StrucDesc -> [[Cat]]` which computes all the sequences of categories that appear along the root-to-leaf paths in the given tree. The order in which the paths appear in the result list does not matter. (Those bigrams . . . they just won't go away . . .)

```
*ContextFree> allPaths sd1
[[S,NP],[S,VP,V],[S,VP,NP]]
*ContextFree> allPaths sd2
[[S,NP],[S,VP,VP,V],[S,VP,PP,P],[S,VP,PP,NP,D],[S,VP,PP,NP,N]]
*ContextFree> allPaths sd3
[[S,NP,D],[S,NP,N],[S,VP,V],[S,VP,S,NP],[S,VP,S,VP,V],[S,VP,S,VP,NP]]
```

I. Write a function `addressesOfNPs :: StrucDesc -> [Address]` which returns a list containing the addresses[2] of all the nodes with category `NP` in the given tree. The order in which the addresses appear in the result list does not matter.

Hint: Let yourself be guided by the way you wrote `numNPs`. In particular, notice that the length of the result of `addressesOfNPs sd` should always be the same as `numNPs sd`; so make sure that this relationship between the two functions holds correctly for the `Leaf` case, and then make sure that you add addresses to lists of addresses "in sync with" the way you incremented numbers in `numNPs`.

```
*ContextFree> addressesOfNPs sd1
[[0],[1,1]]
*ContextFree> addressesOfNPs sd2
[[0],[1,1,1]]
*ContextFree> addressesOfNPs sd3
[[0],[1,1,0],[1,1,1,1]]
*ContextFree> addressesOfNPs sd4
[[0],[1,1,0],[1,1,1,1,1]]
*ContextFree> addressesOfNPs (Leaf V "left")
[]
*ContextFree> addressesOfNPs (Leaf NP "John")
[[]]
```

[2]Recall from the `getSubtree` function in class that the *address* of a node in a given tree is a list of integers that describes how to get from the root of the tree to that node. For the trees that we are dealing with here, which are at most binary branching, the only integers that can appear in addresses are zero and one. A zero is an instruction to step downwards in the tree from whatever node you are currently at to the leftmost daughter of that node (which may or may not be the only daughter); a one is an instruction to step downwards to the second-from-the-left daughter (i.e. for our trees that are at most binary branching, the right daughter). To find the node at a given address, we start at the root node and follow the "instructions" in the address

J. Write a function `replace :: StrucDesc -> Address -> StrucDesc -> StrucDesc` such that the result of evaluating `replace oldTree addr newPart` is a structural description that is like `oldTree` but with `newPart` in place of whatever was previously at the position described by `addr`. The result should be `undefined` if the address does not pick out a valid node in the tree given as the first argument (following the same criteria as `getSubtree`).

```
*ContextFree> replace sd1 [1,1] (Leaf NP "John")
Binary S (Leaf NP "John") (Binary VP (Leaf V "left") (Leaf NP "John"))
*ContextFree> replace sd1 [0] (Leaf NP "Mary")
Binary S (Leaf NP "Mary") (Binary VP (Leaf V "left") (Leaf NP "Mary"))
*ContextFree> replace sd1 [0] (Binary NP (Leaf D "the") (Leaf N "cat"))
Binary S (Binary NP (Leaf D "the") (Leaf N "cat"))
         (Binary VP (Leaf V "left") (Leaf NP "Mary"))
*ContextFree> brackets (replace sd2 [1,1,0] (Leaf P "without"))
"[John [[left] [without [a cat]]]]"
```

Optional extras (no course credit, just for fun)

- Try writing a function `generate :: [GrammarRule] -> Numb -> [StrucDesc]` for these grammars, analogous to the `generate` functions from the last two assignments but where the number argument specifies the maximum *depth* of the trees generated. This is the natural way to understand the number of "steps" by which we extend an initial collection of well-formed SDs, but things are a little bit trickier here because some SDs are built up out of *two* smaller SDs. What this means is that we need an `extendByOne` function whose type is not `[GrammarRule] -> StrucDesc -> [StrucDesc]`, but rather is `[GrammarRule] -> [StrucDesc] -> [StrucDesc]` — it looks at *all* the well-formed SDs that we have built up so far, and creates SDs that are one step deeper than we previously had. This change in type means in turn that the particular placements of `concat` and `map` differ a bit from the previous versions too. But in broad terms, the same approach as before is the right way to go.
- While we can re-use the same basic approach to writing `generate` as we used with FSAs (and bigram grammars), notice that the question of how to write a function `parse :: [GrammarRule] -> Cat -> [String] -> [StrucDesc]` for context-free grammars looks different, and hard. In the case of an FSA, the list of strings we're given to parse corresponds in a certain significant way to the sequence of transitions that make up an SD — but this is no longer true with tree-shaped SDs.

in order. The address of the root node is `[]`, because to get there you take no steps.