LING185A, Assignment #5

Due date: Wed. 2/14/2018

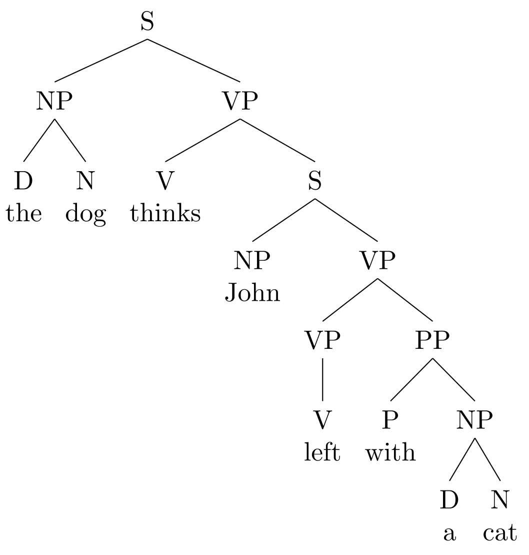
Download `ContextFree_Stub.hs` from the course website, and rename it to `ContextFree.hs`.[1] This file contains the code for working with context free grammars that we looked at in class. You will need to submit a modified version of this file. There are `undefined` stubs for each of the questions below.

Background: Tree addresses

The *address* of a node in a given tree is a list of integers that identifies the position of the node in the tree (in much the same way as an *index* identifies the position of an element in a list). You can think of a node's address as something 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 in order. The address of the root node is `[]`, because to get there you "go nowhere" from the root.

For example, in the tree structure shown here, which corresponds to `sd2`:

- The node at address `[0]` is the NP node covering the words 'the dog'.
- The node at address `[1]` is the VP node covering the words 'thinks . . . cat'.
- The node at address `[0,1]` is the N node covering the word 'dog'.
- The node at address `[1,0]` is the V node covering the word 'thinks'.
- The node at address `[1,1,1,1]` is the PP node.
- The node at address `[1,1,1,0]` is the VP node covering the word 'left'.
- The node at address `[1,1,1,0,0]` is the V node covering the word 'left'.
- The node at address `[]` is the S node at the root.



Notice that not all addresses are valid for a given tree. For example, in the tree above the address

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

[1,1,1,1,1,1,1] is invalid, and so is the address [1,1,1,0,1]. And so is any address that contains any integer other than 0 and 1, for this tree and for any of the trees that we're considering here, but of course some addresses with larger numbers will be valid if we consider trees with more than two branches.

For working with addresses, it's sometimes handy to "case split" on some integer n that appears in the address like this, where the underscore provides an "otherwise" case:

```
case n of {0 -> ...; 1 -> ...; _ -> ...}
```

For example:[2]

```
Prelude> let n = 0 in case n of {0 -> "go left"; 1 -> "go right"; _ -> "invalid"}
"go left"
Prelude> let n = 1 in case n of {0 -> "go left"; 1 -> "go right"; _ -> "invalid"}
"go right"
Prelude> let n = 2 in case n of {0 -> "go left"; 1 -> "go right"; _ -> "invalid"}
"invalid"
Prelude> let n = 3 in case n of {0 -> "go left"; 1 -> "go right"; _ -> "invalid"}
"invalid"
```

Over to you . . .

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). Remember that since strings are just lists (of characters), you can use `++` to concatenate them, e.g. `"[" ++ "hello"` produces `"[hello"`.

```
*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]]]]]]"
*ContextFree> brackets sd5
"[[dogs [[dogs] chase]] chase]"
```

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> labeledBrackets sd1
"[S John [VP left Mary]]"
*ContextFree> labeledBrackets sd2
"[S John [VP [VP left] [PP with [NP a cat]]]]"
```

[2]This messes up the usual nice correspondence between `case`-expressions and the structure of types, i.e. the "options" that we set up when we declare a type. An alternative way of doing things that would keep this correspondence intact would be to define a type like this: `data Address = Left Address | Right Address | Here`. This would also bring out the connection between addresses (which serve as indexes into trees) and natural numbers (which serve as indexes into lists).

```
*ContextFree> labeledBrackets sd3
"[S [NP the dog] [VP thinks [S John [VP left Mary]]]]"
*ContextFree> labeledBrackets sd4
"[S [NP the dog] [VP thinks [S John [VP [VP left] [PP with [NP a cat]]]]]]"
*ContextFree> labeledBrackets sd5
"[S [NP dogs [RC [NP dogs] chase]] chase]"
```

C. Write a function `numNPs`, with type `StrucDesc -> Int`, which computes the number of nodes with category `NP` in the given structural description. Remember that 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 sd5
2
*ContextFree> numNPs (Binary VP sd1 sd2)
4
*ContextFree> numNPs (Binary NP sd1 sd2)
5
```

D. 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`. (We might think of this as a very simplistic measure of "how bad" a sentence is predicted to sound to a native speaker with the given grammar.)

```
*ContextFree> numViolations grammar1 sd1
0
*ContextFree> numViolations grammar1 sd5
8
*ContextFree> numViolations grammar1 (Binary S sd1 sd2)
1
*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
*ContextFree> numViolations grammar2 sd1
5
*ContextFree> numViolations grammar2 sd5
0
```

E. 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!]]]]"
```

F. 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 sd5
[S,NP,RC,NP,N]
*ContextFree> longestPath (Binary S (Leaf NP "foo") (Leaf VP "bar"))
[S,NP]
```

G. 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 sd5
[[S,NP,N],[S,NP,RC,NP,N],[S,NP,RC,TV],[S,IV]]
```

H. Write a function `addressesOfNPs :: StrucDesc -> [Address]` which returns a list containing the addresses of all the nodes with category `NP` in the given tree. `Address` is just a type synonym for `[Int]`. 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 sd5
[[0],[0,1,0]]
```

```
*ContextFree> addressesOfNPs (Leaf NP "John")
[[]]
```

I. Write a function `ccommand :: Address -> Address -> Bool` such that `ccommand addr1 addr2` is `True` iff the node at `addr1` in a tree c-commands the node at `addr2` in that tree.[3] You can assume for this question that the addresses will only contain `0` and `1`. (Notice that we don't care which actual tree structure these nodes might be a part of; the addresses themselves contain all the information we need to decide whether they stand in the c-command relation.)

```
*ContextFree> ccommand [0] [0,1,0,1,0]
False
*ContextFree> ccommand [0] [1,1,0,1,0]
True
*ContextFree> ccommand [1,0] [1,1,1,0,1,0]
True
*ContextFree> ccommand [1,0] [1,0,1,0,1,0]
False
*ContextFree> ccommand [1,1,1,1,0] [1,1,1,1,1,0]
True
*ContextFree> ccommand [1,1,1,1,1,0] [1,1,1,1,1,0]
False
*ContextFree> ccommand [1,1,1,1,1,0] [1,1,1,1,0]
False
```

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 the subtree whose root node is at the position described by `addr`. The result should be `oldTree` unchanged if the address does not pick out a valid node in the tree given as the first argument.

```
*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 [0,0] (Leaf NP "Mary")
Binary S (Leaf NP "John") (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> labeledBrackets (replace sd2 [1,1,0] (Leaf P "without"))
"[S John [VP [VP left] [PP without [NP a cat]]]]"
```

K. Write a function `move :: Address -> StrucDesc -> StrucDesc` which "fronts" the constituent at the given address in the given tree. The fronted constituent should be left-adjoined at the root of the tree, i.e. the root node of the resulting tree should have the same label as the input tree, and it should have the fronted constituent as its left daughter. The right daughter should be like the input tree, but with a "trace" in place of the moved constituent: for our purposes, we can take a trace to be a leaf node which has the same category as the moved constituent and simply has the string `"t"`.

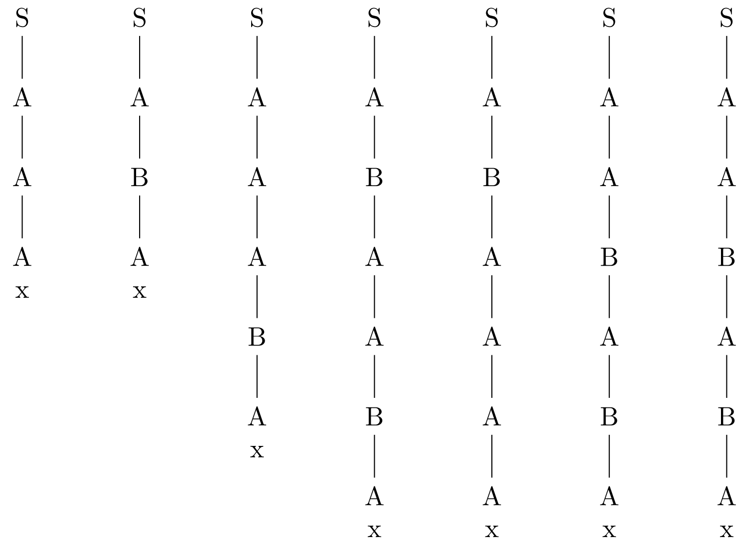
```
*ContextFree> move [1,1] sd1
Binary S (Leaf NP "Mary") (Binary S (Leaf NP "John") (Binary VP (Leaf V "left") (Leaf NP "t")))
*ContextFree> move [1,1] sd2
```

[3]Recall that X c-commands Y iff (i) X does not dominate Y, (ii) Y does not dominate X, and (iii) the lowest node dominating X also dominates Y (where domination is not reflexive). This definition is not particularly helpful for writing the required function though — it's better to think about the geometry of trees directly.

```
Binary S (Binary PP (Leaf P "with") (Binary NP (Leaf D "a") (Leaf N "cat")))
         (Binary S (Leaf NP "John") (Binary VP (Unary VP (Leaf V "left")) (Leaf PP "t")))
*ContextFree> labeledBrackets (move [1,1] sd2)
"[S [PP with [NP a cat]] [S John [VP [VP left] t]]]"
*ContextFree> labeledBrackets (move [1,1,0] sd3)
"[S John [S [NP the dog] [VP thinks [S t [VP left Mary]]]]]"
```

Things to think about (no course credit, just for fun)

- Consider the context-free grammar that has just these four rules: S → A, A → B, B → A and A → x. Let's take S to be the start symbol. Then the grammar generates trees like the ones below. Is it possible to write a context-free grammar which generates only the trees that conform to this pattern but also have *exactly one* B node in them? Where have we seen this point before?



- 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. Instead of an `extendByOne` function whose type is `[GrammarRule] -> StrucDesc -> [StrucDesc]`, we need something that looks at *all* the well-formed SDs that we have built up so far (rather than just one of them), and creates SDs that are one step deeper than we previously had. 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.