1 Option 1: Pushdown automata and sentence-processing

This project involves writing some Haskell code implementing pushdown automata (PDAs) and the bottom-up, top-down and left-corner conversions from CFGs to PDAs. Your paper will describe the more complicated parts of your implementation, and discuss what this implementation can show us about the suitability of CFGs as models of natural language syntax.

Download `PDA_Stub.hs` and rename it to `PDA.hs`. This file contains all the code you need to get started. You'll submit a modified version of this file, as well as your paper as a PDF file.

1.1 Setup and starter code

1.1.1 Pushdown automata

As a reminder, here's the definition of a pushdown automaton we saw earlier in the course:

(1) A pushdown automaton (PDA) is a five-tuple $(\Gamma, \Sigma, I, F, \Delta)$ where:
- Γ is the *stack alphabet*;
- Σ is the *(surface) alphabet*;
- $I \subseteq \Gamma^*$ is a *finite* set of initial stack contents;
- $F \subseteq \Gamma^*$ is a *finite* set of ending stack contents; and
- $\Delta \subseteq \Gamma^* \times (\Sigma \cup \{\epsilon\}) \times \Gamma^*$ is a *finite* set of transitions.

And here's the definition of one particular PDA, which generates the language $\{\mathrm{a}^n\mathrm{b}^n \mid n \geq 0\}$.

(2) $\Gamma = \{\mathrm{X}, \mathrm{Y}, \mathrm{A}\}$
$\Sigma = \{\mathrm{a}, \mathrm{b}\}$
$I = \{\mathrm{X}\}$
$F = \{\mathrm{Y}\}$
$\Delta = \{({}_\uparrow\mathrm{X}, \mathrm{a}, {}_\uparrow\mathrm{XA}), ({}_\uparrow\mathrm{X}, \epsilon, {}_\uparrow\mathrm{Y}), ({}_\uparrow\mathrm{YA}, \mathrm{b}, {}_\uparrow\mathrm{Y})\}$

I've defined `PDA st sy` as a type for representing a PDA where the stack symbols have type `st`[1] and the surface symbols have type `sy`. To keep things a bit more manageable, the stack alphabet and the surface alphabet, Γ and Σ, are not represented explicitly as separate components — we can kind of think of these as being implicitly specified by the type parameters `st` and `sy` — so the Haskell type is actually just a triple, corresponding to (I, F, Δ). Since the places where a PDA can start and end (i.e. the members of the sets I and F) are *strings* of stack symbols, each one of these places has type `[st]`, and so the first two components of the tuple have type `[[st]]`. Since we allow ϵ transitions, the type of a transition is `([st], Maybe sy, [st])`.

For an example, take a look at `pda_anbn`, which represents the PDA in (2). Notice that in (2) I've written the strings of stack symbols with the top of the stack on the *left*: this is different from the way we wrote stacks at one point, but is in line with how we ended up writing stacks for top-down and left-corner PDAs. The main reason for doing things this way in the code is that it puts the top of the stack as the "outermost" element, which you can get your hands on immediately: if a list has the form `x:rest`, then `x` is the element on the top of the stack.

There's also `pda_ambig`, which is a silly little PDA that generates all non-empty strings of 'a's, and does so in such a way that any string of 'a's is "ambiguous", in the sense that it has two different transition sequences. This will be useful for illustrating something relevant later.

As well as `PDA st sy`, I've defined `TransitionSeq st sy` as a type for representing a *transition sequence* taken by a PDA whose stack symbols have type `st` and whose surface symbols have type `sy`. Such a sequence might be zero transitions long, in which case it represents just "starting" with a particular string of symbols on the stack; or it might be n transitions long, for some $n \geq 1$, in which case it consists of some sequence of

[1] You can think of `st` as a kind of a pun on "stack" and "state".

$n-1$ transitions, along with a pair consisting of the additional/nth transition and the string of symbols that is on the stack after taking that transition. (This last transition is the "outermost" one in the sequence, so this type is a bit like the type for snoc lists that we saw earlier in the course.)

For an example, take a look at `seq_aabb`, which represents the transition sequence by which `pda_anbn` accepts the string `"aabb"`. Just for convenience, I've set things up so that when ghci wants to show you something of type `TransitionSeq st sy` (or a list of such things), it uses a friendly format which looks a bit like the tables we sometimes write on paper:

```
*PDA> Start "X" ::: (("X", Just 'a', "XA"), "XA")
---                     "X"
("X",Just 'a',"XA")     "XA"
*PDA> seq_aabb
---                     "X"
("X",Just 'a',"XA")     "XA"
("X",Just 'a',"XA")     "XAA"
("X",Nothing,"Y")       "YAA"
("YA",Just 'b',"Y")     "YA"
("YA",Just 'b',"Y")     "Y"
```

This is convenient but it's important to remember that it's only a matter of how transition sequences are *displayed*: the definition you see in the source file is still the definition of what transition sequences actually *are*, and this is the only thing that matters when it comes to working with them in your code.

I've provided a function `resultStack` that simply pulls out the stack that a transition sequence "ends at". Notice that there always is such a stack, even if the sequence is zero transitions long, because of the way the base case of this recursive type is set up.

1.1.2 Context-free grammars

The types `CFG nt sy` and `RewriteRule nt sy` are similar to what we saw earlier in the course, with two small differences. First, we're not representing the set of nonterminal symbols and the set of terminal symbols explicitly as separate components (just like we left out the Γ and Σ components of a PDA), so a CFG is just a pair, consisting of a set of initial nonterminal symbols and a set of rewrite rules. Second, rather than Chomsky Normal Form, here we allow *one or more* (but not zero) nonterminal symbols on the right-hand side of an `NTRule`. We do this by representing the right-hand side of a rule as a pair `(nt,[nt])`, where the first component of the pair is the first nonterminal on the right-hand side of the rule (the obligatory one, we might say), and the second component of the pair is a list consisting of everything after the first one (possibly empty, of course). If you think about it, this corresponds exactly to leaving out the "empty list" possibility for the right-hand side of the rule: a non-empty list, remember, just *is* an element along with a list. This format might seem a bit clunky at first but will turn out to be quite convenient. (It sits nicely with *left-corner* processing.)

As examples, I've defined `cfg_eng`, which is a simple English-like CFG that is very similar to the ones we were working with in Week 7, and `cfg_anbn`, which is a small CFG that generates $\{\mathrm{a}^n\mathrm{b}^n \mid n \geq 1\}$.

1.2 Warm up exercises: PDA transitions [5% of the project grade]

Write a function

```
tryTransition :: (Eq st) => Transition st sy -> [st] -> Maybe [st]
```

such that `tryTransition tr s` "tries to" take the transition `tr`, given current stack contents `s`. If `tr` is not applicable to `s`, then the result should be `Nothing`; if `tr` is applicable to `s`, the result should be `Just s'`, where `s'` is the new stack contents. Notice that the surface symbol specified by a transition (the middle element of the triple) does not play any role here. Here are some examples of how it should behave.

```
*PDA> tryTransition ("X", Just 'a', "XA") "XAAAAA"
```

```
Just "XAAAAAA"
*PDA> tryTransition ("X", Just 'a', "XA") "X"
Just "XA"
*PDA> tryTransition ("X", Just 'a', "XA") ""
Nothing
*PDA> tryTransition ("X", Just 'a', "XA") "AX"
Nothing
*PDA> tryTransition ([0], Nothing, [5]) [0,1,2,3]
Just [5,1,2,3]
*PDA> tryTransition ([0], Nothing, [5,5,5]) [0,1,2,3]
Just [5,5,5,1,2,3]
*PDA> tryTransition ([0,1,2], Nothing, [5]) [0,1,2,3]
Just [5,3]
*PDA> tryTransition ([0,1,2], Nothing, [5]) [0,1]
Nothing
```

Now, use `tryTransition` to write a function

```
checkSequence :: (Eq st) => TransitionSeq st sy -> Bool
```

which checks whether a transition sequence is "internally consistent", i.e. whether taking the specified sequence of transitions indeed produce the specified changes in stack contents. Notice that you're not checking for whether anything is allowed by any particular PDA here. And again, the surface symbols are not playing any role.

```
*PDA> checkSequence seq_aabb
True
*PDA> checkSequence (seq_aabb ::: (("YA", Just 'b', "Y"), "Y"))
False
*PDA> checkSequence (seq_aabb ::: (("Y", Just 'b', "Y"), "Y"))
True
*PDA> checkSequence (seq_aabb ::: (("Y", Just 'b', "ZZZ"), "ZZZ"))
True
*PDA> checkSequence (Start [0,1,2,3] ::: (([0,1,2], Nothing, [5]), [5,3]))
True
*PDA> checkSequence (Start [0,1] ::: (([0,1,2], Nothing, [5]), [5,3]))
False
*PDA> checkSequence (Start [0,1])
True
```

1.3 Finding transition-sequences in a PDA [25% of the project grade]

Here's the meaty part ...

Write a function

```
completeSequences :: (Eq st, Eq sy) =>
    PDA st sy -> ([st] -> Bool) -> TransitionSeq st sy -> [sy] -> [TransitionSeq st sy]
```

so that the list returned by `completeSequences pda fn seq w` contains all the transition sequences `seq'` which extend `seq` in such a way that

- if `checkSequence` returns `True` on `seq`, then it returns `True` on `seq'` too;
- the transitions added to `seq` are all allowed by `pda`;
- the transitions added to `seq` produce the surface symbols `w` (in order);

- the transitions added to `seq` only proceed via stacks on which `fn` returns `True`; and
- `resultStack seq'` is one of `pda`'s allowed ending stacks.

That sounds complicated, but the idea is that if `seq` is the initial portion of a transition sequence allowed by `pda`, and that initial portion produces the surface string `v`, then this function should find completions of `seq` that represent ways `pda` can generate the string `v++w`. The one additional requirement — this will seem strange for now, but will be useful later — is that we are restricting attention to completions that only use the stack in accord with a constraint specified by the function `fn`: at all points along the new part of the transition sequence, the stack must be something on which `fn` returns `True`. The things on which `fn` returns `False` are "out of bounds", in effect. Notice that you do not need to check whether stacks already visited/reached in `seq` satisfy this `fn` requirement; you just need to make sure that the additional steps only lead to (and only proceed via) stacks that satisfy this requirement.

To get started with this, it will be easiest to just ignore the requirement specified by `fn` for a while; don't try to enforce this requirement until you have the rest of it working. At this initial stage, the function should give the same results as the eventual finished version will need to in cases where the `fn` argument is `\x -> True`, i.e. nothing is out of bounds. Here are some examples.

```
*PDA> completeSequences pda_ambig (\x -> True) (Start [0]) "aaa"

---                           [0]
([0],Just 'a',[0])            [0]
([0],Just 'a',[1])            [1]
([1],Just 'a',[3])            [3]

---                           [0]
([0],Just 'a',[0])            [0]
([0],Just 'a',[2])            [2]
([2],Just 'a',[3])            [3]

*PDA> completeSequences pda_anbn (\x -> True) (Start "X" ::: (("X",Just 'a',"XA"),"XA")) "b"

---                           "X"
("X",Just 'a',"XA")           "XA"
("X",Nothing,"Y")             "YA"
("YA",Just 'b',"Y")           "Y"

*PDA> completeSequences pda_anbn (\x -> True) (Start "X" ::: (("X",Just 'a',"XA"),"XA")) "abb"

---                           "X"
("X",Just 'a',"XA")           "XA"
("X",Just 'a',"XA")           "XAA"
("X",Nothing,"Y")             "YAA"
("YA",Just 'b',"Y")           "YA"
("YA",Just 'b',"Y")           "Y"

*PDA> completeSequences pda_anbn (\x -> True) (Start "XAAA") "bbb"

---                           "XAAA"
("X",Nothing,"Y")             "YAAA"
("YA",Just 'b',"Y")           "YAA"
("YA",Just 'b',"Y")           "YA"
("YA",Just 'b',"Y")           "Y"

*PDA> completeSequences pda_anbn (\x -> True) (Start "XAAA") "bb"
[]
```

As long as you get `completeSequences` working correctly in these cases, ignoring the out-of-bounds restriction, you'll be able to move on to the next section and make a decent fist of it — you'll run into a small problem at a certain point, where you will want to be able to impose a restriction on the way this search for transition sequences works, but not being able to do so won't mean you're at a complete dead end. You'll be able to get a good overall grade for the project if you explain the situation in your paper, and describe how the out-of-bounds restriction *would* be useful below, even if you weren't able to implement it.

If you do manage to get the out-of-bounds restriction working, here are some examples of how things should work.

```
*PDA> completeSequences pda_ambig (\s -> not (elem 1 s)) (Start [0]) "aaa"

---                          [0]
([0],Just 'a',[0])           [0]
([0],Just 'a',[2])           [2]
([2],Just 'a',[3])           [3]

*PDA> completeSequences pda_anbn (\s -> not (elem 'X' s)) (Start "XAA" ::: (("X",Just 'a',"XA"),"XAAA")) "bbb"

---                          "XAA"
("X",Just 'a',"XA")          "XAAA"
("X",Nothing,"Y")            "YAAA"
("YA",Just 'b',"Y")          "YAA"
("YA",Just 'b',"Y")          "YA"
("YA",Just 'b',"Y")          "Y"

*PDA> completeSequences pda_anbn (\s -> not (elem 'X' s)) (Start "XAA") "abbb"
[]
```

As a small extra step, use `completeSequences` to write a function

`pdaRecognize :: (Eq st, Eq sy) => PDA st sy -> ([st] -> Bool) -> [sy] -> [TransitionSeq st sy]`

such that `pdaRecognize pda fn w` finds all the transition sequences by which `pda` generates/accepts the string `w`, with `fn` specifying an out-of-bounds restriction in the same way as above for `completeSequences`.

```
*PDA> pdaRecognize pda_ambig (\x -> True) "aaaa"

---                          [0]
([0],Just 'a',[0])           [0]
([0],Just 'a',[0])           [0]
([0],Just 'a',[1])           [1]
([1],Just 'a',[3])           [3]

---                          [0]
([0],Just 'a',[0])           [0]
([0],Just 'a',[0])           [0]
([0],Just 'a',[2])           [2]
([2],Just 'a',[3])           [3]

*PDA> pdaRecognize pda_anbn (\x -> True) "aaabbb"

---                          "X"
("X",Just 'a',"XA")          "XA"
("X",Just 'a',"XA")          "XAA"
("X",Just 'a',"XA")          "XAAA"
("X",Nothing,"Y")            "YAAA"
("YA",Just 'b',"Y")          "YAA"
("YA",Just 'b',"Y")          "YA"
("YA",Just 'b',"Y")          "Y"
```

```
*PDA> pdaRecognize pda_anbn (\s -> length s < 4) "aaabbb"
[]
```

1.4 Conversions from a CFG to a PDA [20% of the project grade]

Now the fun starts . . .

1.4.1 Bottom-up

Write a function

```
bottomUpPDA :: CFG nt sy -> PDA nt sy
```

which converts the given CFG to a PDA that generates the same string language, using the bottom-up conversion schema.

Then use `bottomUpPDA` and `pdaRecognize` to write a function

```
bottomUpRecognize :: (Eq nt, Eq sy) => CFG nt sy -> [sy] -> [TransitionSeq nt sy]
```

that finds all the ways a bottom-up PDA derived from the given CFG can process the given string.

Here are some examples using `cfg_anbn`.

```
*PDA> bottomUpPDA cfg_anbn
([[]],[[0]],[([2,0,1],Nothing,[0]),([2,1],Nothing,[0]),([],Just 'a',[1]),([],Just 'b',[2])])
*PDA> bottomUpRecognize cfg_anbn "aabb"

---                           []
([],Just 'a',[1])             [1]
([],Just 'a',[1])             [1,1]
([],Just 'b',[2])             [2,1,1]
([2,1],Nothing,[0])           [0,1]
([],Just 'b',[2])             [2,0,1]
([2,0,1],Nothing,[0])         [0]

*PDA> bottomUpRecognize cfg_anbn "aab"
[]
```

For working with `cfg_eng`, the predefined function `words` is handy: its type is `String -> [String]`, and it splits a string up at spaces.

```
*PDA> words "the baby saw the boy"
["the","baby","saw","the","boy"]
*PDA> bottomUpRecognize cfg_eng (words "the baby saw the boy")

---                           []
([],Just "the",[D])           [D]
([],Just "baby",[N])          [N,D]
([N,D],Nothing,[NP])          [NP]
([],Just "saw",[V])           [V,NP]
([],Just "the",[D])           [D,V,NP]
([],Just "boy",[N])           [N,D,V,NP]
([N,D],Nothing,[NP])          [NP,V,NP]
([NP,V],Nothing,[VP])         [VP,NP]
([VP,NP],Nothing,[S])         [S]

*PDA> bottomUpRecognize cfg_eng (words "the actor the boy the baby saw met won")
```

```
---                            []
([],Just "the",[D])            [D]
([],Just "actor",[N])          [N,D]
([],Just "the",[D])            [D,N,D]
([],Just "boy",[N])            [N,D,N,D]
([],Just "the",[D])            [D,N,D,N,D]
([],Just "baby",[N])           [N,D,N,D,N,D]
([N,D],Nothing,[NP])           [NP,N,D,N,D]
([],Just "saw",[V])            [V,NP,N,D,N,D]
([V,NP],Nothing,[ORC])         [ORC,N,D,N,D]
([ORC,N,D],Nothing,[NP])       [NP,N,D]
([],Just "met",[V])            [V,NP,N,D]
([V,NP],Nothing,[ORC])         [ORC,N,D]
([ORC,N,D],Nothing,[NP])       [NP]
([],Just "won",[V])            [V,NP]
([V],Nothing,[VP])             [VP,NP]
([VP,NP],Nothing,[S])          [S]

*PDA> length (bottomUpRecognize cfg_eng (words "John saw the baby with the telescope"))
2
```

1.4.2 Top-down

Write a function

    topDownPDA :: CFG nt sy -> PDA nt sy

which converts the given CFG to a PDA that generates the same string language, using the top-down conversion schema.

Then use `topDownPDA` and `pdaRecognize` to write a function

    topDownRecognize :: (Eq nt, Eq sy) => CFG nt sy -> [sy] -> [TransitionSeq nt sy]

that finds all the ways a top-down PDA derived from the given CFG can process the given string.

```
*PDA> topDownPDA cfg_anbn
([[0]],[[]],[([0],Nothing,[1,0,2]),([0],Nothing,[1,2]),([1],Just 'a',[]),([2],Just 'b',[])])
*PDA> topDownRecognize cfg_anbn "aabb"

---                            [0]
([0],Nothing,[1,0,2])          [1,0,2]
([1],Just 'a',[])              [0,2]
([0],Nothing,[1,2])            [1,2,2]
([1],Just 'a',[])              [2,2]
([2],Just 'b',[])              [2]
([2],Just 'b',[])              []

*PDA> topDownRecognize cfg_anbn "aab"
[]
```

```
*PDA> topDownRecognize cfg_eng (words "the baby saw the boy")

---                            [S]
([S],Nothing,[NP,VP])          [NP,VP]
([NP],Nothing,[D,N])           [D,N,VP]
([D],Just "the",[])            [N,VP]
```

```
([N],Just "baby",[])          [VP]
([VP],Nothing,[V,NP])         [V,NP]
([V],Just "saw",[])           [NP]
([NP],Nothing,[D,N])          [D,N]
([D],Just "the",[])           [N]
([N],Just "boy",[])           []

*PDA> topDownRecognize cfg_eng (words "the actor the boy the baby saw met won")

---                           [S]
([S],Nothing,[NP,VP])         [NP,VP]
([NP],Nothing,[D,N,ORC])      [D,N,ORC,VP]
([D],Just "the",[])           [N,ORC,VP]
([N],Just "actor",[])         [ORC,VP]
([ORC],Nothing,[NP,V])        [NP,V,VP]
([NP],Nothing,[D,N,ORC])      [D,N,ORC,V,VP]
([D],Just "the",[])           [N,ORC,V,VP]
([N],Just "boy",[])           [ORC,V,VP]
([ORC],Nothing,[NP,V])        [NP,V,V,VP]
([NP],Nothing,[D,N])          [D,N,V,V,VP]
([D],Just "the",[])           [N,V,V,VP]
([N],Just "baby",[])          [V,V,VP]
([V],Just "saw",[])           [V,VP]
([V],Just "met",[])           [VP]
([VP],Nothing,[V])            [V]
([V],Just "won",[])           []

*PDA> length (topDownRecognize cfg_eng (words "John saw the baby with the telescope"))
2
```

1.4.3 Left-corner

Write a function

```
leftCornerPDA :: CFG nt sy -> PDA (LCS nt) sy
```

which converts the given CFG to a PDA that generates the same string language, using the left-corner conversion schema.

Notice that the type of the stack symbols in the resulting PDA is not simply `nt` here, because this PDA's stack alphabet needs to have two symbols for each nonterminal in the CFG — one to represent constituents already found bottom-up, and one to represent constituents predicted top-down. To look after this, the type of the stack symbols in the resulting PDA is `LCS nt`, whose definition (already provided for you) is simply:

```
data LCS nt = Predicted nt | Found nt deriving Eq
```

On paper, we wrote the bottom-up/found symbols "as is" (e.g. N, VP), and wrote the top-down/predicted symbols with a bar (e.g. $\overline{\text{N}}$, $\overline{\text{VP}}$). To make things look a bit more familiar (as well as more compact), I've set up this type so that ghci displays these in a similar way, with an asterisk in place of the bar.

```
*PDA> Found D
D
*PDA> Predicted S
S*
*PDA> [Found D, Predicted S]
[D,S*]
```

But as with the transition-sequences above, it's important to remember that this is *only* a matter of how these things are displayed for you, and has no bearing on how you should write code to work with these symbols.

Then use `leftCornerPDA` and `pdaRecognize` to write a function

```
leftCornerRecognize :: (Eq nt, Eq sy) => CFG nt sy -> [sy] -> [TransitionSeq (LCS nt) sy]
```

that finds all the ways a left-corner PDA derived from the given CFG can process the given string.

```
*PDA> leftCornerRecognize cfg_anbn "aaabbb"

---                           [0*]
([],Just 'a',[1])             [1,0*]
([1,0*],Nothing,[0*,2*])      [0*,2*]
([],Just 'a',[1])             [1,0*,2*]
([1,0*],Nothing,[0*,2*])      [0*,2*,2*]
([],Just 'a',[1])             [1,0*,2*,2*]
([1,0*],Nothing,[2*])         [2*,2*,2*]
([2*],Just 'b',[])            [2*,2*]
([2*],Just 'b',[])            [2*]
([2*],Just 'b',[])            []

*PDA> leftCornerRecognize cfg_eng (words "the baby saw the boy")

---                           [S*]
([],Just "the",[D])           [D,S*]
([D],Nothing,[N*,NP])         [N*,NP,S*]
([N*],Just "baby",[])         [NP,S*]
([NP,S*],Nothing,[VP*])       [VP*]
([],Just "saw",[V])           [V,VP*]
([V,VP*],Nothing,[NP*])       [NP*]
([],Just "the",[D])           [D,NP*]
([D,NP*],Nothing,[N*])        [N*]
([N*],Just "boy",[])          []

*PDA> length (leftCornerRecognize cfg_eng (words "John saw the baby with the telescope"))
2
```

1.5 Paper [50% of the project grade]

Make sure you've read the general guidelines provided on page 1.

Of course, when it comes to describing your implementation, you don't need to repeat information from the instructions given on the preceding pages. The most significant part to describe will probably be your implementation of the `completeSequences` function.

Here are some specific issues that you should address in discussing what you've done:

- Do your `bottomUpRecognize`, `topDownRecognize` and `leftCornerRecognize` functions correctly identify all of the possible analyses of a string?
- Do these functions show the memory usage (stack depth) growing in the ways you expected as a function of the length of the string, for left-embedding, right-embedding and center-embedding patterns? Can you say something about how these three kinds of structures relate to the difference between PDAs and FSAs?

- What do these memory usage patterns let us conclude about the suitability of CFGs as a model of human grammars? What should we conclude about the status of sentences like 'the actor the boy the baby saw met won'? Explain.
- What use(s) did you find for the out-of-bounds restriction on the search for transition sequences in a PDA? Explain where this was necessary and why.
- What were the most challenging parts to implement? What were the crucial insights or realizations that allowed you to get back on track when you got stuck, or allowed you to fix an initially-mysterious bug?
- Is there some way in which this exercise has deepened or sharpened your understanding of the relationships between FSAs, CFGs and PDAs? Explain.

Some other things you might like to think about, perhaps in the "loose ends" or "next steps" category:

- Notice that `cfg_anbn` does not generate quite the same set of strings as `pda_anbn`: the way we've set up CFGs here, there is no way for a CFG to generate the empty string. To do this we would need to allow rewrite rules of the form 'A $\rightarrow \epsilon$'. How would allowing rules of this form complicate the schemas for constructing PDAs for a CFG? Is this more complicated than having ϵ-transitions in a PDA, which we already allow? Could the out-of-bounds restriction be useful?
- How might you go about implementing something that constructed the parse trees corresponding to the transition sequences found by a bottom-up and/or top-down and/or left-corner PDA? Which of these three conversion schemas would this task be easier and harder for?
- When there are multiple possible transition sequences for a string, which ones come before others in the list that is returned by functions like `pdaRecognize`? How could their order in this list be interpreted as saying something about the decisions humans make in the course of processing sentences? How does this relate to the different kinds of transitions (e.g. SHIFT vs. REDUCE, PREDICT vs. MATCH) that your CFG-to-PDA conversion functions construct?