Take-home final exam

Kevin Liang Ling 185A Due: 09/17/2024, 11:59 PM PDT

Instructions: This is a take home exam. You are free to refer to handouts, assignments, quizlets, and your notes as you work through it. You can look elsewhere if you like, but be warned that this will probably not be the most efficient way to find what you need. **You must complete your final exam entirely on your own. Please do not discuss the exam with anyone except Kevin.**

Total point: 60 + 5 extra credits

Download `FinalUtilities.hs` and `Final_Stub.hs`, save them in the same directory, and rename the latter to `Final.hs`. (Please use this name exactly.) For the questions below you'll add some code to this file, and then submit your modified version. You should also submit a PDF file with your written responses to Section 3, and Section 4.2 (and Section 2.2-C). Do not modify or submit `FinalUtilities.hs`.

1 "Snoc lists" and Forward values (20 points)

Notice that the built-in Haskell list type, whose definition looks approximately like this:

```
data [a] = [] | a :  [a] deriving Show
```

Recall that `deriving Show` throws an instance of the just-defined type into Show typeclass; and it is just a way to get `ghci` to print expressions of this type.

arbitrarily makes the *leftmost* element of a list the *outermost* element (i.e. the one you get your hands on straight away when using a `case` statement).

Sometimes it's convenient to be able to work with lists the other way around, i.e. treating the *rightmost* element as the outermost element, the one that gets "peeled off" first when we analyze the list. For this we can define our own type like this (already in `FinalUtilities.hs`):

```
data SnocList a = ESL | (SnocList a) :::  a deriving Show
```

These are called "snoc lists," because— well, the first kind of lists are called "cons lists." A snoc list is either empty (i.e. `ESL` for "empty snoc list") or comprised of a snoc list and then an element; in the latter case, the snoc operator ::: is what holds them together.

So we could represent the word "hello" as a snoc list as follows:

```
sl :::  SnocList Char
sl = ((((ESL :::  'h') :::  'e') :::  'l') :::  'l') :::  'o'
```

Then, if we were to deconstruct this using a case statement like the one below, the variable `x` on the last line would refer to the character `'o'`:[1]

```
case sl of
ESL -> ...
rest:::x -> ...
```

A. Write a function `addToFront :: a -> SnocList a -> SnocList a` so that `addToFront x l` returns a snoc list that is like `l` but has an additional occurrence of `x` as its leftmost element.

2 points

```
*Final> addToFront 2 (((ESL ::: 3) ::: 4) ::: 5)
(((ESL ::: 2) ::: 3) ::: 4) ::: 5
```

[1]Don't mix up `'h'` and `"h"`: `'h'` is a character (type `Char`), but `"h"` is an abbreviation for `['h']`, i.e. a cons-list-of- characters (type `[Char]`) whose length happens to be one. Similarly, `"hello"` is an abbreviation for `['h','e','l','l','o']`, which of course is just an abbreviation for `'h':('e':('l':('l':('o':[]))))`. But `'hello'` is gibberish.

```
*Final> addToFront 'x' ((ESL ::: 'y') ::: 'z')
((ESL ::: 'x') ::: 'y') ::: 'z'
*Final> addToFront 'x' ESL
ESL ::: 'x'
*Final> addToFront False (ESL ::: True)
(ESL ::: False) ::: True
```

B. Write a function `toSnoc :: [a] -> SnocList a` that produces the snoc list corresponding to the given "normal list" (i.e. cons list). 3 points

```
*Final> toSnoc ['h','e','l','l','o']
((((ESL ::: 'h') ::: 'e') ::: 'l') ::: 'l') ::: 'o'
*Final> toSnoc [3,4,5]
((ESL ::: 3) ::: 4) ::: 5
*Final> toSnoc [True]
ESL ::: True
```

C. Write a function `forward :: (Eq a) => Automaton State a -> SnocList a -> State -> Bool` which computes forward values. That is, `forward m w q` should evaluate to `True` iff there's a way to get from an initial state of the automaton `m` to the state `q` that produces the symbols of `w`. For this question, follow the definition in (13) on the lecture 2 handout. Looking at the implementation of `backward` might also be helpful. 6 points

```
*Final> forward fsa_handout4 ((ESL ::: C) ::: V) 43
False
*Final> forward fsa_handout4 ((ESL ::: C) ::: V) 42
True
*Final> forward fsa_handout4 (toSnoc [C,V]) 42
True
*Final> map (forward fsa_handout4 (toSnoc [C,V,V])) [40,41,42,43]
[True,False,True,True]
*Final> map (forward fsa_handout5 (toSnoc [C,V])) [1,2,3]
[True,True,False]
*Final> map (forward fsa_handout5 (toSnoc [C,V,C])) [1,2,3]
[True,False,True]
*Final> map (forward fsa_handout5 (toSnoc [C,V,C,C])) [1,2,3]
[True,False,False]
*Final> map (forward fsa_handout5 (toSnoc [C,V,C,C,V])) [1,2,3]
[True,True,False]
```

D. Write a function `generatesViaForward :: (Eq a) => Automaton State a -> SnocList a -> Bool` which checks whether the given automaton generates the given string of symbols. This should produce the same results as the existing function generates, but you should use your forward function to do it. (See (11) on the handout for lecture 2.) 4 points

```
*Final> generatesViaForward fsa_handout5 (toSnoc [C,V,C,C])
False
*Final> generatesViaForward fsa_handout5 (toSnoc [C,V,C])
True
```

E. Now, write a function `forwardGeneric :: (Semiring v) => GenericAutomaton st sy v -> SnocList sy -> st -> v` which computes generalized forward values, following equation (29) on the lecture 6 handout. So for a given string str and a given state q, this function computes, in a semiring-general way, a semiring-general way, the value that an automaton associates with "getting to q" in a way that produces str. 3 points

```
*Final> forwardGeneric gfsa5 (toSnoc "VC") 2
True
*Final> forwardGeneric gfsa5 (toSnoc "VC") 3
```

```
False
*Final> forwardGeneric gfsa23 (toSnoc "VC") 3
0.0
*Final> forwardGeneric gfsa23 (toSnoc "VC") 2
0.1
*Final> forwardGeneric gfsa6 (toSnoc "CVCV") 1
1.0
*Final> forwardGeneric gfsa6 (toSnoc "CVCV") 2
0.0
```

F. Now, write a function `fViaForward :: (Semiring v) => GenericAutomaton st sy v -> SnocList sy -> v` which computes values of the function f_M, using equation (31b) on the lecture 6 handout. 2 points

```
*Final> fViaForward gfsa5 (toSnoc "VCV")
True
*Final> fViaForward gfsa5 (toSnoc "CVCV")
True
*Final> fViaForward gfsa6 (toSnoc "VCV")
0.9
*Final> fViaForward gfsa6 (toSnoc "CVCV")
1.0
*Final> fViaForward gfsa23 (toSnoc "VCV")
9.000000000000001e-3
*Final> fViaForward gfsa23 (toSnoc "CVCV")
1.1250000000000001e-2
```

2 Strictly-local grammars and FSAs (20 points)

Recall from lecture 8, like an FSA, a $2-$SLG generates a set of strings over some alphabet. Formally, a $2-$SLG is a four-tuple (Σ, I, F, T), where

- Σ is the alphabet of symbols,
- I is a subset of Σ, specifying the allowable starting symbols,
- F is a subset of Σ, specifying the allowable final symbols, and
- T is a subset of $\Sigma \times \Sigma$, i.e. a set of pairs of symbols, specifying the allowable two-symbol sequences (or allowable "bigrams").

Notice one important difference between SLGs and FSAs: SLGs have no states!

2.1 Recognizing strings generated by an SLG

An SLG (Σ, I, F, T) generates a string of n symbols $x_1 x_2 ... x_n$ iff:

- $x_1 \in I$, and
- for all $i \in \{2, ..., n\}$, $(x_{i-1}, x_i) \in T$, and
- $x_n \in F$

Notice that by this definition, there is no way for an SLG to generate the empty string.[2]

For any type `sy` that our chosen symbols belong to, we can straightforwardly represent an SLG in Haskell as a tuple with the type `([sy], [sy], [sy], [(sy,sy)])`, where the four components specify the alphabet, the starting symbols, the final symbols and the allowable bigrams, respectively.

```
type SLG sy = ([sy], [sy], [sy], [(sy,sy)])
```

For example, the SLG in 1 (with `SegmentCV` as its symbol type) generates all strings consisting of one or more Cs followed by one or more Vs; and the SLG in 2 (with `Int` as its symbol type) generates all strings built out of the symbols 1, 2 and 3 which do not have adjacent occurrences of 2 and 3 (in either order). These two SLGs are defined for you with the names `slg1` and `slg2`.

1. `([C,V], [C], [V], [(C,C),(C,V),(V,V)])`
2. `([1,2,3], [1,2,3], [1,2,3], [(1,1),(2,2),(3,3),(1,2),(2,1),(1,3),(3,1)])`

A. Write a function `generatesSLG :: (Eq sy) => SLG sy -> [sy] -> Bool` which checks whether the given string of symbols is generated by the given SLG. 5 points

There are a few different ways to do this, but one way is to write a recursive helper function analogous to `backward` for FSAs (called, say, `backwardSLG`), which then allows a non-recursive implementation of `generatesSLG`, analogous to `generates` for FSAs.

Here are some examples of how it should behave:

```
*Final> generatesSLG slg1 [C,C,V]
True
*Final> generatesSLG slg1 [C,V]
True
*Final> generatesSLG slg1 [V]
```

[2]This is slightly non-standard. Actually, the definition of strictly-local grammars we introduced in this class slightly diverge from the usual definitions in the literature. The standard definition includes special start-of-string and end-of-string markers as components of bigrams, which makes it possible to generate the empty string.

```
False
*Final> generatesSLG slg1 [V,C]
False
*Final> generatesSLG slg2 [1]
True
*Final> generatesSLG slg2 [1,2,1,2,1,3]
True
*Final> generatesSLG slg2 [1,2,1,2,1,3,2]
False
*Final> generatesSLG slg2 []
False
```

Of course, it should work for all SLGs, not just the two defined above:

```
*Final> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","ha"]
True
*Final> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa"]
False
*Final> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","mwa"]
False
*Final> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","ha","
    ha"]
True
```

2.2 Design SLGs and FSAs

5 points

A. Suppose a language L in which the well-formed words are those that satisfy the following requirements:

- The initial syllable has primary stress
- Words contain exactly one syllable with primary stress.
- Words contain at least one syllable.

Actually, Latvian exhibits this kind of stress pattern.

Assume $\Sigma = \{\acute{\sigma}, \sigma\}$, where $\acute{\sigma}$ is a syllable with primary stress, σ is an unstressed syllable. Thus, the grammar accepts only sequences of syllables that conform to the stress pattern defined above.

- Some accepted strings: $\acute{\sigma}$, $\acute{\sigma}\sigma$, $\acute{\sigma}\sigma\sigma$, etc.
- Some rejected strings: $\sigma\sigma$, $\acute{\sigma}\acute{\sigma}$, σ, etc.

I've defined the Haskell type `SyllableTypes` as follows:

```
data SyllableTypes = Stressed | Unstressed deriving (Eq, Show)
```

Define a grammar `slgStress :: SLG SyllableTypes` that generates all and only the well-formed words of L.

Don't worry if your `generatesSLG` does not work; the grading for this question won't be based on your `generatesSLG` implementation.

```
*Final> generatesSLG slgStress [Stressed]
True
*Final> generatesSLG slgStress [Stressed, Unstressed]
True
*Final> generatesSLG slgStress [Stressed, Unstressed, Unstressed]
True
*Final> generatesSLG slgStress [Unstressed, Unstressed]
False
*Final> generatesSLG slgStress [Stressed, Stressed]
```

```
False
*Final> generatesSLG slgStress [Unstressed]
False
*Final> generatesSLG slgStress [Unstressed, Stressed]
False
```

B. Define an FSA in Haskell as `fsaHarmony :: Automaton State SegmentPKIU`. This FSA should have {P, K, I, U, MB} as its alphabet and enforce a simple kind of *vowel harmony*: if we interpret 'MB' as a morpheme boundary, all the vowels within a morpheme must be identical to each other.[3] Any strings built out of this alphabet are allowed as long as they satisfy this requirement. This includes some strange ones such as those that contain two adjacent "morpheme boundaries," or have a "morpheme boundary" at the beginning or end—but never mind, the goal is just to isolate out the vowel harmony requirement itself. It should behave like this: 5 points

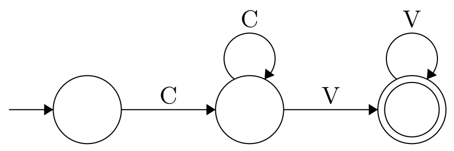
```
*Final> generates fsaHarmony [P,K,I,K,MB,U,P,U]
True
*Final> generates fsaHarmony [P,K,I,K,U,P,U]
False
*Final> generates fsaHarmony [K,I,P,I]
True
*Final> generates fsaHarmony [K,P,P,P]
True
*Final> generates fsaHarmony [K,I,P,U]
False
*Final> generates fsaHarmony [K,I,MB,P,U]
True
*Final> generates fsaHarmony [MB,MB,K,MB,P]
True
```

C. [optional] This vowel harmony pattern is not strictly local. Explain why. Extra 5 points

2.3 Conversion to FSAs

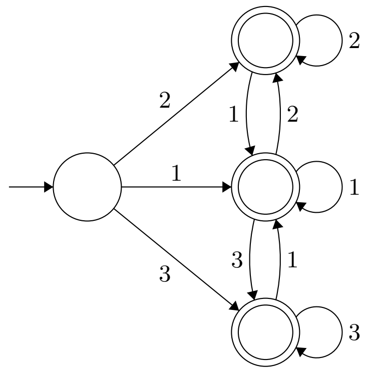
It turns out that any stringset that can be generated by an SLG can also be generated by some FSA. We know this because for any SLG there is a mechanical recipe for constructing an FSA that generates exactly the same stringset as that SLG. Your task here is to figure out what that recipe is. The following examples should give you enough to figure it out: applying the recipe to the `slg1` produces the FSA in 1 and applying the recipe to 2 produces the FSA in 2.

1. `slg1`:=`([C,V], [C], [V], [(C,C),(C,V),(V,V)])`



[3]The concept of a morpheme doesn't actually play any role here; it's just a boundary of some sort that splits the string into "chunks" that the vowel harmony constraint applies to.

2. `slg2:=([1,2,3], [1,2,3], [1,2,3], [(1,1),(2,2),(3,3),(1,2),(2,1),(1,3),(3,1)])`



Hints: Recall the relationship between the "bucketings" of strings and the states of an FSA. Notice that when L is the stringset generated by an SLG, $u \equiv_L v$ if the strings u and v end with the same symbol. Notice also that the SLG in 2 has *three* symbols, and the equivalent FSA in 2 has *four* states.

A. Write a function `slgToFSA` that takes a SLG as input, and produces an equivalent FSA. 5 points

```
data ConstructedState sy = ExtraState | StateForSymbol sy
```

So the type `ConstructedState sy` has one value in it for every value of the type `sy`, plus the additional special value `ExtraState`. The result then is that `slgToFSA` will have this type:

```
slgToFSA :: SLG sy -> Automaton (ConstructedState sy) sy
```

When this is working properly, the result of evaluating `slgToFSA slg1` should correspond to the FSA in 1, and the result of evaluating `slgToFSA slg2` should correspond to the FSA in 2. And these results can be used with the existing `generates` function for FSAs. That is, `generates (slgToFSA g) x` should give the same result as `generatesSLG g x`, for any SLG `g` and any string `x`.

```
*Final> generates (slgToFSA slg1) [C,C,V]
True
*Final> generates (slgToFSA slg1) [V,C]
False
*Final> generates (slgToFSA slg2) [1,2,1,2,1,3]
True
*Final> generates (slgToFSA slg2) []
False
*Final> generates (slgToFSA (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")])) ["mwa
    ","mwa"]
False
*Final> generates (slgToFSA (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")])) ["mwa
    ","ha","ha"]
True
```

3 More on stack-based parsing (10 points)

S → NP VP	N → baby, boy, actor, award, boss
S → WHILE S S	NP → Mary, John
NP → NP POSS N	V → met, saw, won, cried, watched
NP → (D) N (PP) (SRC) (ORC)	D → the
VP → V (NP) (PP)	P → on, in, with
PP → P NP	THAT → that
SRC → THAT VP	POSS → 's
ORC → NP V	WHILE → while

Let's suppose that Martians, to the best of our knowledge, have the grammar given above as their mental grammar. Then we discover certain new kinds of sentences that Martians produce and/or judge to be acceptable sentences of their language, that have not been noticed before. Here are some examples of this new kind of sentence:

1. (a) John said loudly Mary won
   (b) Mary said quietly John 's boss won
   (c) the boss said slowly the actor met Mary

So now the question is, what new rules should we add to the grammar above in order to account for these new kinds of sentences? We will consider two hypotheses. Note that no matter which of these two hypotheses we adopt, the new grammar will generate exactly the same set of sentences. So there is *no way to distinguish between these two hypotheses on the basis of which sentences the two grammars generate.*

- **Hypothesis #1:**
  VP → SAID ADV S
  SAID → said
  ADV → loudly, quietly, slowly

- **Hypothesis #2:**
  VP → X S
  X → SAID ADV
  SAID → said
  ADV → loudly, quietly, slowly

We do know, however, that these Martians use **bottom-up parsing**. And it's generally accepted that a memory limitation is responsible for the fact that, although Martians find (2a) to be a perfectly unremarkable sentence, (2b) makes their heads spin. So although (2b), like (2a), is generated by the rules in our current grammar and is therefore assumed to in fact be *grammatical* for Martians, there is something else going wrong when they try to process (2b). To record this observed difference I'll put an asterisk next to those sentences that produce a "something-went-wrong" reaction from the Martians.

2. (a) John met the boy
   (b) *John met the boy that saw the actor

In order to decide between Hypothesis #1 and Hypothesis #2, a clever linguist devises the following sentences and asks Martians for their judgements of them. As above, the presence or absence of the asterisk indicates the presence or absence of a something-went-wrong reaction.[4]

3. (a) John won
   (b) Mary said quietly John won
   (c) John said slowly Mary said quietly John won
   (d) *John said slowly John said loudly Mary said quietly John won

6 points

A. Show the full sequence of configurations that a bottom-up parser goes through to parse the sentence in (3c) correctly for each hypothesis. **To get full credit, you need to use the format of the table on our lecture handout, and you need to show the transition type and rule used for each step**.

B. How can we use this information to choose between Hypothesis #1 and Hypothesis #2? Explain your reasoning fully. (But there's no need to write out any complete step-by-step transition sequences.) Notice that in this question, unlike the problems on homework 3, the issue is not whether or not larger and larger structures with a certain kind of embedding pattern lead to processing difficulty *at some point* — they clearly do — but rather *at which point* we see the difficulty arising. 4 points

Hint: one thing to notice is that to analyze (2a) and (2b), the only grammar rules used do not involve the extra rules introduced in two hypotheses – so figuring out what makes the difference between (2a) and (2b) is an important first step.

[4]The clever linguist did not bother to find out whether, in the sentences where there are multiple occurrences of 'John', the Martians take these occurrences to necessarily refer to distinct people all called John, as would be the case for many humans. In fact, the clever linguist didn't bother to find out anything at all about how Martians interpret these sentences — it's currently a topic of debate whether the Martian word 'John' is a name at all, or whether it's an adjective meaning something like "pertaining to small quantities of brake fluid" — because their goal was just to decide between Hypothesis #1 and Hypothesis #2, and they couldn't see a way to use information about interpretations to help make that decision.

4 Finite-state tree grammars (10 points)

4.1 Designing FSTAs: Island effects

The basic pattern

You probably learned in syntax class about the fact that wh-movement out of certain kinds of structures is disallowed. We call these structures *islands.* There are many different theories of what makes something an island (e.g. subjacency, phases), but for our purposes here we'll keep things simple: all that matters is that *adjuncts are islands.*

Adjuncts include relative clauses, such as the one indicated in (1), and 'because'-clauses, such as the one indicated in (2).[5]

(1) John likes the person [$_{\text{adjunct}}$ that bought books as a gift]

(2) John laughed [$_{\text{adjunct}}$ because Mary bought books as a gift]

So the generalization that adjuncts are islands explains the issue with movement in the following English examples, where we try to form questions that would be answered with responses 'books' and 'as a gift', respectively. (The *t* in these examples indicates the position from which wh-movement is supposed to originate.)

(3) a. *What does John like the person [$_{\text{adjunct}}$ that bought *t* as a gift]?
b. *Why does John like the person [$_{\text{adjunct}}$ that bought books *t*]?

(4) a. *What did John laugh [$_{\text{adjunct}}$ because Mary bought *t* as a gift]?
b. *Why did John laugh [$_{\text{adjunct}}$ because Mary bought books *t*]?

Interestingly, the configurations that block wh-movement in these English examples also (sometimes) block the non-movement dependency that in-situ wh-words must establish with Q. Here's some relevant data from Mandarin Chinese, showing adjuncts (relative clauses and 'after'-clauses, which we can treat like English 'because'-clauses) in brackets.

(5) *

ni	*zui*	*xihuan*	[*weishenme*	*mai*	*shu*	*de*]	*ren*	*?*
you	most	like	why	buy	book	DE	person	

Intended: 'Why do you like the person that bought books?'

(6) *

ta	[*zai*	*Lisi*	*weishenme*	*mai*	*shu*	*yihou*]	*shengqi*	*le*	*?*
he	at	Lisi	why	buy	book	affter	angry	LE	

Intended: 'Why did he get angry after Lisi bought books?'

And here's something along the same lines from Japanese.

(7) *

Mary-ga	[*John-ni*	*naze*	*hon-o*	*ageta*]	*hito-ni*	*atta*	*no*	*?*
Mary-NOM	John-DAT	why	book-ACC	gave	man-DAT	meet	Q	

Intended: 'Why did Mary meet the man that gave books to John?'

So the constraint that rules out (3) and (4) is arguably not a constraint on *movement*, it's a constraint on the kind of dependency that a wh-word needs to establish with a Q complementizer; this kind of dependency is sometimes reflected on the surface via movement, and sometimes not.[6]

[5]Relative clauses are adjoined to, say, DP, or at least something noun-ish; 'because'-clauses are adjoined to, say, VP or TP, or at least something verb-ish.

[6]I am abstracting away from many details here. The Chinese and Japanese examples above were very carefully hand-picked: the pattern I'm describing holds at least sometimes when the wh-phrase itself is an adjunct, such as 'why', but

Your task

A. Define a finite-state tree automaton `fsta_Island`, with `String` as its symbol type, that (i) enforces the basic dependency pattern between Q and wh-words as `fsta_wh` in Assignment 05, and also (ii) requires that the licensing Q for a wh-word is not separated from it by an adjunct island boundary. (This is similar to the way a reflexive cannot be separated from its binder by a clause boundary.) Again, as in assignment 05, you can use whatever you like as the state type, but you will probably find that you can use the same states you used for `fsta_wh` here. We will use `"**"` as a non-leaf symbol to identify (the root nodes of) adjuncts.

6 points

- Every wh-word must be c-commanded by a Q complementizer.
  A single Q can license multiple wh-words that it c-commands, just like a single negation can license multiple NPIs. When this happens it produces "multiple questions", like the English 'Who ate what?'
- Every Q must c-command at least one wh-word.
  This is different from the pattern with NPIs: a negative element need not c-command an NPI.[7] (I'm leaving aside yes-no questions here.)
- Any leaf node with a string in the list `qWords` is a Q complementizer.
- Any leaf node with a string in the list `whWords` is a wh-word.
- Other leaf nodes are allowed to have any string in the list `plainWords`.
- Non-leaf nodes can now have either the label `"*"` or the label `"**"`. Non-leaf nodes must still have exactly two daughters.
- A constituent whose root node has the label `"**"` is an adjunct.
- If a wh-word is contained within an adjunct, then it can only be licensed by a Q complementizer that is also within that adjunct; every Q complementizer must license at least one wh-word in this way.

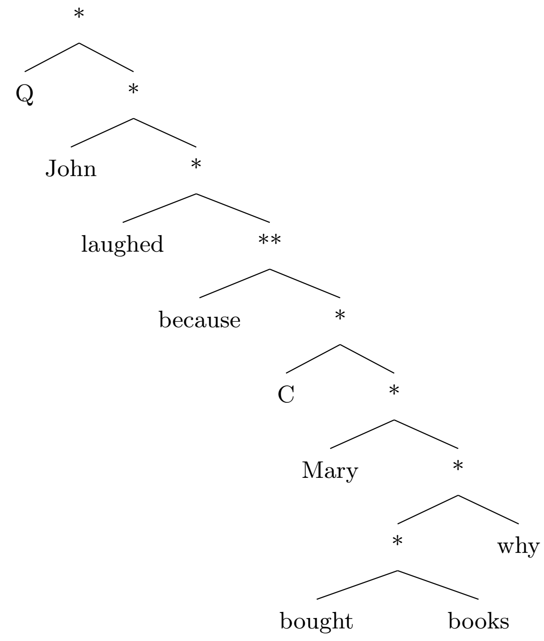
The list of symbols for this tree automaton (i.e. the second component of the four-tuple) should be

`plainWords`, `whWords` and `qWords` are provided in the header of Final_Stub.hs

```
plainWords ++ whWords ++ qWords ++ ["*","**"]
```

The tree for a question based on sentence (2), for example, looks like this:

(8)



generally not with argument wh-phrases, such as 'who' and 'what'.

[7] For example, 'John didn't buy apples' is no worse than 'John didn't buy anything'.

This tree should be rejected because it violates the constraints. This tree is implemented as tree_13 in Haskell.

You can copy `allList`, `under` and `generatesFSTA` you did in assignment 05 to check whether your `fsta_Island` works as expected. Here are some test cases.

The grading for this question will not be based on your previous implementations.

```
*Final> generatesFSTA fsta_Island tree_13
False
*Final> generatesFSTA fsta_Island (Node "*" [Node "Q" [], Node "*" [Node "C" [], Node "why"
    []]])
True
*Final> generatesFSTA fsta_Island (Node "*" [Node "Q" [], Node "**" [Node "C" [], Node "why"
     []]])
False
*Final> generatesFSTA fsta_Island (Node "*" [Node "C" [], Node "**" [Node "Q" [], Node "why"
     []]])
True
```

4.2 Martian long-distance dependencies

One notable feature of the long-distance dependencies we have introduced so far is that *one* licensor can license *multiple* licensees: 4 points

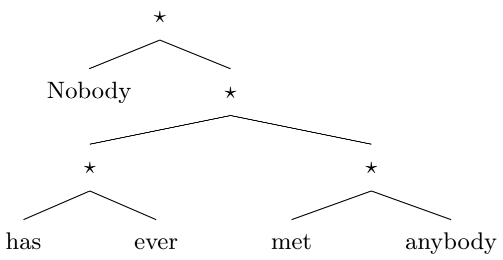
- In the *NPI licensing constraint* we discussed in Week 5, a single negation, such as ‘nobody’ can license multiple NPIs (such as ‘anybody’, ‘ever’).
- In *long-distance wh-dependencies* we saw in Assignment 05 and 4.1 here, a single complementizer Q can license multiple *wh* words.

Recall our simple FSTA for NPI licensing:

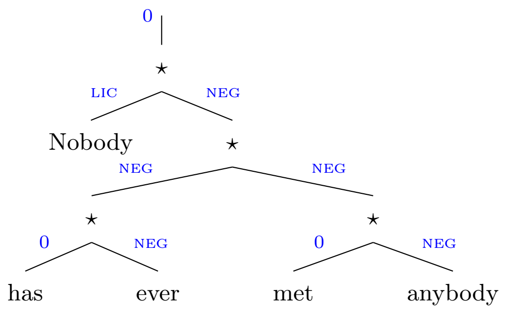
$$
\begin{array}{l}
G_3 = (\{0,\text{LIC},\text{NEG}\},\ \Sigma,\ \{0,\text{LIC}\},\ \Delta)\\
\quad \text{where } \Delta = \begin{array}[t]{llllllll}
\{([\text{NEG},\text{NEG}], & \star, & \text{NEG}), & ([], & \text{anybody}, & \text{NEG}), & \\
([0,\text{NEG}], & \star, & \text{NEG}), & ([], & \text{ever}, & \text{NEG}), & \\
([\text{NEG},0], & \star, & \text{NEG}), & ([], & \text{not}, & \text{LIC}), & \\
([0,0], & \star, & 0), & ([], & \text{nobody}, & \text{LIC}), & \\
([\text{LIC},\text{NEG}], & \star, & 0), & ([], & s, & 0) & \text{for any other } s \in \Sigma - \{\star\}, \\
([\text{LIC},0], & \star, & 0), & & & & \\
([0,\text{LIC}], & \star, & 0), & & & & \\
([\text{LIC},\text{LIC}], & \star, & 0)\} & & & &
\end{array}
\end{array}
$$

For example, the FSTA G_3 accepts the tree in (9), and a possible analysis is shown in (10).

(9) .



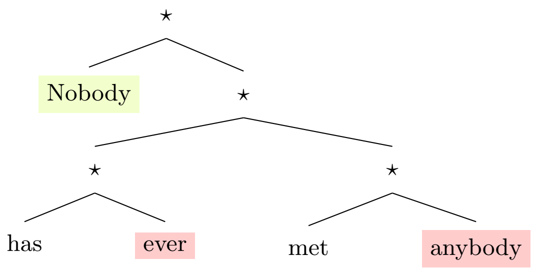
(10) .



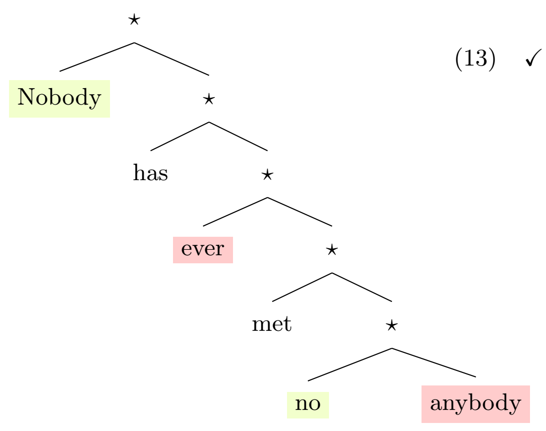
Let us suppose that there is a certain kind of Martian. They speak a language that also shows NPI-licensing constraints in that any licensee must be c-commanded by a licensor, but they require each licensee at different locations to be c-commanded by a different licensor, and therefore, the number of licensors (or negations) must match the number of licensees (or negative polarity items).

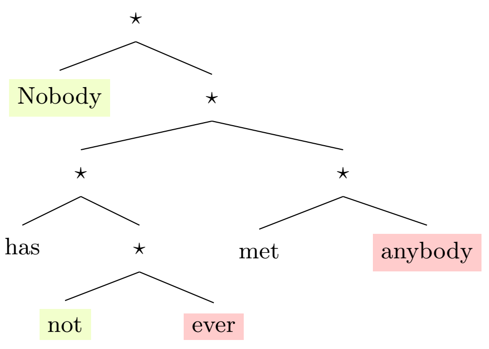
That said, the tree in (9), repeated in (11) should be rejected based on Martian’s grammar: there are two licensees, ‘ever’ and ‘anybody’ but only one licensor, ‘Nobody’.

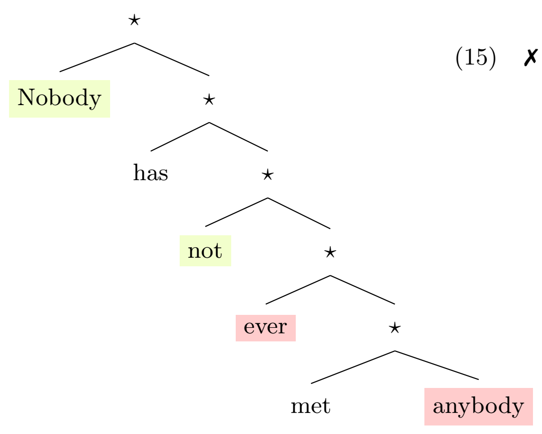
(11) .

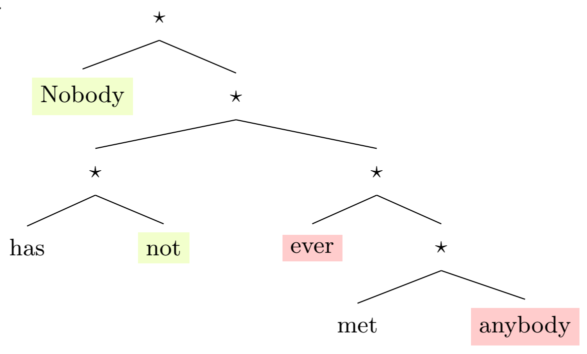


However, Martians instead would mark (12), (13) and (14) as well-formed trees: they are accepted because now there are two licensors (‘nobody’, and ‘not’), and two licensees (‘ever’, and ‘anybody’) and they are in c-commanding relations in the tree (licensee c-commended by licensor).

(12) ✓


(13) ✓


(14) ✓


(15) ✗


On the other hand, the Martian grammar should reject the tree in (15): even though there are two licensors for two licensees, the licensor ‘not’ doesn’t c-command any licensees and thus doesn’t participate in resolving dependencies.

Moreover, the Martian grammar does not restrict the number of negative polarity items and negations appearing in the trees. They can accept forms with 25 negative polarity items as long as there are also 25 negations as licensors such that they form c-commanding relations.

Your task: Is there a way to modify G_3 so that it generates the set of trees that satisfy the Martian NPI licensing constraint? If so, describe how this can be done. (You don’t need to write out the fully modified grammar, but describe the additions/deletions/changes to G_3 precisely.) If not, explain why this is not possible.