Assignment #8

Due date: Thu. 3/4/2021, 12 noon

Download `TreeGrammars.hs` and `Assignment08_Stub.hs` from the CCLE site, save them in the same directory, and rename the latter to `Assignment08.hs`. (Please use this name exactly.) For the questions below you'll add some code to this file, and then submit your modified version. Do not modify or submit `TreeGrammars.hs`.

Background and reminders

Have a look at `TreeGrammars.hs`. Things to notice:

- There are definitions of `Tree sy` and `Automaton st sy` that directly correspond to definitions we saw in class.
- You can see some concrete example of how we represent certain trees in `t7`, `t14` and `t20`. Notice that every node has a list of daughters, even if it's empty.
- Notice that *every tree* has the form `Node x ts`, for some symbol `x` and some (possibly empty) list of trees `ts`. There is no other case to consider; there is no vertical bar in the definition of `Tree sy`.

You can ignore the starter code in `Assignment08.hs` to begin; it will be explained below in section 3.

1 Warm-ups: Working with trees

These don't have anything to do with tree automata yet.

A. Write a function `count :: (Eq a) => a -> Tree a -> Int` which returns the number of occurrences of the given symbol in the given tree.

```
*Assignment08> count 'a' t7
2
*Assignment08> count 'b' t7
3
*Assignment08> count 'c' t7
1
*Assignment08> count 'a' (Node 'a' [t7,t7,t7,t7])
9
*Assignment08> count "that" t20
1
*Assignment08> count "*" t20
5
```

B. Write a function `leftEdge :: Tree a -> [a]` which returns the sequence of symbols we find by following a root-to-leaf path downwards through given tree, taking the leftmost branch at each point. (Remember that `"bca"` is just `['b','c','a']`.)

```
*Assignment08> leftEdge t7
"bca"
*Assignment08> leftEdge t14
"abba"
*Assignment08> leftEdge t20
["*","*","that"]
*Assignment08> leftEdge (Node 3 [])
[3]
```

2 Computing "under" values

Your task here will be to implement the definitions in section 2.3 of the class handout. This is very similar to things we've seen before with forward values, backward values, and inside values.

C. A useful piece of the puzzle will be to first write a helper function `allLists :: Int -> [a] -> [[a]]` which produces all lists of the given length, made out of elements of the given list. The order in which these lists appear in the result does not matter. Remember that `[]` has length zero. You can assume that the `Int` argument is non-negative. (In principle, it would be nice to use the `Numb` type here; instead, you'll have to `case`-split on the type `Bool`.)

```
*Assignment08> allLists 2 ['x','y']
["xx","xy","yx","yy"]
*Assignment08> allLists 2 ['x','y','z']
["xx","xy","xz","yx","yy","yz","zx","zy","zz"]
*Assignment08> allLists 3 [0,1]
[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]
*Assignment08> length (allLists 5 [0,1])
32
*Assignment08> length (allLists 10 [0,1])
1024
```

D. Write a function

```
under :: (Eq st, Eq sy) => Automaton st sy -> Tree sy -> st -> Bool
```

which implements the definition in (11) on the class handout. Notice that we are representing automata using lists of states and lists of transitions — not functions from states/transitions to booleans — so you will need to *check for membership* in these lists using `elem`, rather than *applying* corresponding functions.

```
*Assignment08> under fsta_even t14 Even
True
*Assignment08> under fsta_even (Node 'b' [t14,t14]) Odd
False
*Assignment08> under fsta_npi t20 NegOK
True
*Assignment08> under fsta_npi t20 Neg
False
*Assignment08> under fsta_npi t20 LicNeg
False
```

E. Write a function

```
generates :: (Eq st, Eq sy) => Automaton st sy -> Tree sy -> Bool
```

which checks whether the given finite-state tree automaton generates the given tree.

```
*Assignment08> generates fsta_npi t20
True
*Assignment08> generates fsta_npi (Node "*" [Node "anything" [], t20])
False
*Assignment08> generates fsta_even t14
True
*Assignment08> generates fsta_even (Node 'a' [t14])
False
```

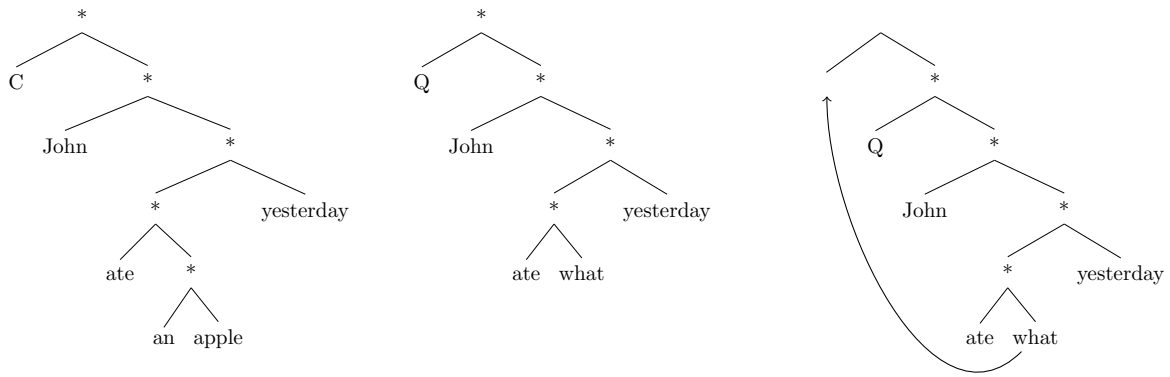
3 Wh-in-situ dependencies

3.1 The basic pattern

In syntax classes you've probably talked about "wh-in-situ" languages (such as Japanese, Mandarin, Turkish, Hindi, and many others), where wh-words that move to the front of a question in languages like English instead stay in their base position. So in a language which is wh-in-situ but is otherwise like English — let's call it Whinglish — a question formed from (1a) by asking about the object position would look almost the same as (1a), as shown in (1b), rather than (1c) where we see the effect of wh-movement.[1]

(1) a. John ate an apple yesterday
b. John ate what yesterday
c. what John ate yesterday

Recall that in (1c), 'who' has moved to the specifier of the complementizer projection near the top of the tree. This complementizer makes the utterance a question and attracts a wh-word or wh-phrase. The complementizer in (1c), however, is different, and does not have these properties. We might say that the complementizer in (1c) has a +Q feature whereas the complementizer in (1a) has a −Q feature, or something like that, but just to keep things simple let's write Q for the question-forming complementizer in (1c) and write C for the "plain" complementizer in (1a). In wh-in-situ languages, we usually assume that there is a question-forming complementizer Q in questions like (1b) as well. So the trees for the three sentences above look like this:



Evidence for the idea that there's a distinguished Q complementizer comes from the fact that it's pronounced overtly in many languages. For example, in Japanese it's pronounced 'no'; see (2), corresponding to (1a) and (1b). The C complementizer in the declarative is unpronounced, like in English.[2] The Q complementizer

[1] Leaving aside the separate issue of subject-auxiliary inversion (i.e. T-to-C movement) in English questions.
[2] Or perhaps we should say it's pronounced as ϵ.

must co-occur with a wh-word, and vice-versa, as shown in (3). (The difference between head-initial order in English and head-final order in Japanese is irrelevant for us here — the point is that the word order is the same in the declarative sentence and the question, just with the addition of the question-marker at the edge of the clause.)

(2) a. John-wa ringo-o tabeta
John-TOP apple-ACC ate
'John ate an apple'

b. John-wa nani-o tabeta no
John-TOP what-ACC ate Q
'What did John eat?'

(3) a. * John-wa ringo-o tabeta no
John-TOP apple-ACC ate Q

b. * John-wa nani-o tabeta
John-TOP what-ACC ate

So there is a *dependency* between the Q complementizer and the wh-word, in the Japanese (2b) and in the Whinglish (1b) — the top part of a clause that has a Q complementizer can't go with the bottom part of a clause that doesn't have a wh-word — even if the establishment of that dependency does not lead to "movement" as it does in the English (1c). We'll leave aside the issue of movement here and just consider the dependency itself (which, of course, is *part of* what happens in English wh-questions too, even if it's not the whole story).

From this point of view, the dependency between a wh-word and a Q complementizer is somewhat similar to the dependency between an NPI and its licensing negative element, which we discussed in class. Specifically, here's a complete set of rules for well-formed trees:

(4) a. 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?'.

b. 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.[3]
(I'm leaving aside yes-no questions here.)

c. Any leaf node with a string in the list `qWords` is a Q complementizer.

d. Any leaf node with a string in the list `whWords` is a wh-word.

e. Other leaf nodes are allowed to have any string in the list `plainWords`.

f. Non-leaf nodes must have the label `"*"`, and must have exactly two daughters.

Your task here is to define `fsta_wh1` to be a finite-state tree automaton, with `String` as its symbol type, that enforces these rules. You can choose whatever you like as the state type; you might like to define a custom type for this purpose, like we did for `fsta_npi`. (If you do this, you will need to include `deriving (Eq,Show)`.)

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

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

Here are some examples of how `fsta_wh1` should behave:

```
*Assignment08> generates fsta_wh1 tree_1a
True
*Assignment08> generates fsta_wh1 tree_1b
True
*Assignment08> generates fsta_wh1 tree_3a
False
*Assignment08> generates fsta_wh1 tree_3b
```

[3] For example, 'Nobody bought apples' is no worse than 'Nobody bought anything'.

```
False
*Assignment08> generates fsta_wh1 (mrg (lf "Q") (lf "what"))
True
*Assignment08> generates fsta_wh1 (mrg (lf "Q") (lf "Mary"))
False
*Assignment08> generates fsta_wh1 (mrg (lf "C") (lf "what"))
False
*Assignment08> generates fsta_wh1 (mrg (lf "Q") (mrg (lf "what") (lf "why")))
True
*Assignment08> generates fsta_wh1 (mrg (lf "C") (mrg (lf "what") (lf "why")))
False
*Assignment08> generates fsta_wh1 (mrg (lf "Q") (mrg (lf "John") (lf "what")))
True
*Assignment08> generates fsta_wh1 (mrg (mrg (lf "Q") (lf "John")) (lf "what"))
False
```

3.2 Island effects

You probably also learned, in syntax classes, about the way 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 (5), and 'because'-clauses, such as the one indicated in (6).[4]

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

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

So the generalization that adjuncts are islands identifies what's wrong with the movement in the following English examples, where we try to form the questions that would be answered by 'books' or 'as a gift'.

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

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

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.

(9) * ni zui xihuan [weishenme mai shu de] ren ?
you most like why buy book DE person
Why do you like the person that bought books?

(10) * ta [zai Lisi weishenme mai shu yihou] shengqi le ?
he at Lisi why buy book after angry LE
Why did he get angry after Lisi bought books?

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

[4] 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.

(11) * Mary-ga [John-ni naze hon-o ageta] hito-ni atta no ?
Mary-NOM John-DAT why book-ACC gave man-DAT meet Q
Why did Mary meet the man that gave books to John?

So the constraint that rules out (7) and (8) 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.[5]

Your task here is to define a finite-state tree automaton `fsta_wh2`, with `String` as its symbol type, that (i) enforces the basic dependency pattern between Q and wh-words as `fsta_wh1` does, 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, 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_wh1` here. We will use `"**"` as a non-leaf symbol to identify (the root nodes of) adjuncts. Specifically:

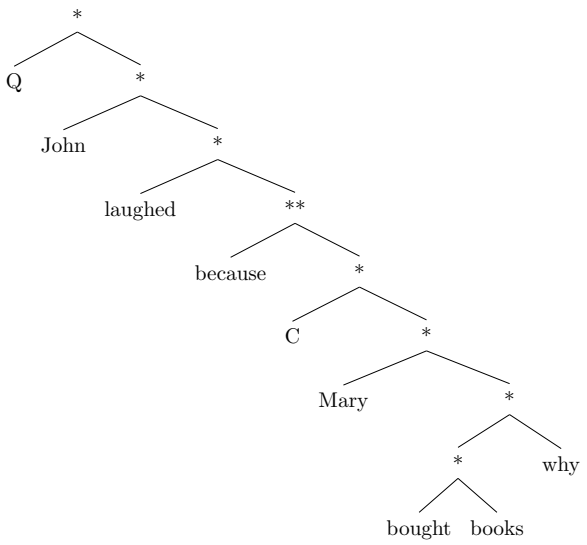
(12) a. All of the rules from (4) above still apply, *except that* non-leaf nodes can now have either the label `"*"` or the label `"**"`. Non-leaf nodes must still have exactly two daughters.
b. A constituent whose root node has the label `"**"` is an adjunct.
c. If a wh-word is contained within an 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 ++ qWords ++ ["*","**"]
```

The tree for a question based on sentence (6), for example, looks like this:

(13)



This needs to be rejected by `fsta_wh2`. It is defined for you as `tree_13`. But all that actually matters is the structural relationship between Q, the wh-word, and the adjunct constituent.

```
*Assignment08> generates fsta_wh2 tree_13
False
*Assignment08> generates fsta_wh2 (Node "*" [Node "Q" [], Node "**" [Node "C" [], Node "why" []]])
```

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

```
False
*Assignment08> generates fsta_wh2 (Node "*" [Node "Q" [], Node "*" [Node "C" [], Node "why" []]])
True
*Assignment08> generates fsta_wh2 (Node "*" [Node "Q" [], Node "**" [Node "C" [], Node "why" []]])
False
*Assignment08> generates fsta_wh2 (Node "*" [Node "C" [], Node "**" [Node "Q" [], Node "why" []]])
True
```