Ling 185A: Assignment 8

Dylan Bumford, Winter 2024

due: Fri, Mar 15

1 Background and reminders

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

- Top-down tree automata and functions are designated with `_td`. These are included for reference and as examples. All the exercises will work with bottom-up tree automata, which are designated without a suffix. So `FTA q`, for instance, is a bottom-up finite tree automaton.
- You can see some concrete examples of encoded trees in `t7`, `t14` and `t20`.
- Remember that *every node has a list of daughters*, though if it's a leaf node that list will be empty. So every tree has the form `Node x ts`, for some symbol `x` and some (possibly empty) list of trees `ts`. To compute, say, the number of nodes in a tree, we could write this:

```
nodes :: Tree a -> Int
nodes (Node x ts) = 1 + sum (map nodes ts)
```

- There are some little convenience functions `lf` (leaf), and `mrg` (binary merge when the root label doesn't matter) to make building trees slightly less tedious. These should be self-evident.

1.1 Warm-ups: Working with trees

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

1. Write a function `count` which returns the number of occurrences of a string in a tree.

```
count :: String -> Tree String -> Int
count x (Node y ts) = undefined
```

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

2. 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.

```
leftEdge :: Tree a -> [a]
leftEdge (Node y ts) = undefined
```

```
ghci> leftEdge t7
["b","c","a"]
ghci> leftEdge t14
["a","b","b","a"]
ghci> leftEdge t20
[".",".","that"]
ghci> leftEdge (Node 3 [])
[3]
```

1.2 Finite Tree Automata Parsing

3. Write a function to parse a tree using a bottom-up tree automaton. This should return the list of states that a machine could end up in after consuming the tree from the leaves upward, using the transitions sanctioned by the machine. You may wish to consider the top down parser `walkFTA_td` defined in class (and in `FTA.hs`) for comparison.

```haskell
walkFTA :: Eq q => [Transition q] -> Tree String -> [q]
walkFTA delta (Node c cs) = undefined
```

```
ghci> walkFTA (getDelta evenAsFTA) t14
[E]
ghci> walkFTA (getDelta polarityFTA) t20
[Meh]
```

4. Write a function to check whether a given `FTA` can generate a given tree. Remember that in order for the tree to be grammatical, there must be at least one tree-shaped path through the machine that ends in an acceptable final state.

```haskell
generates :: Eq q => FTA q -> Tree String -> Bool
generates m t = undefined
```

```
ghci> generates polarityFTA t20
True
ghci> generates polarityFTA (Node "." [Node "anything" [], t20])
False
ghci> generates evenAsFTA t14
True
ghci> generates evenAsFTA (Node "a" [t14])
False
```

2 Wh-in-situ dependencies

2.1 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 otherwise like English — let's call it Whenglish — 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.

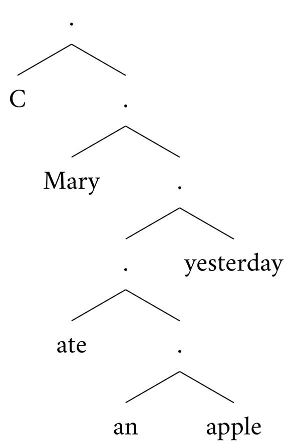
(1) a. Mary ate an apple yesterday

b. Mary ate what yesterday

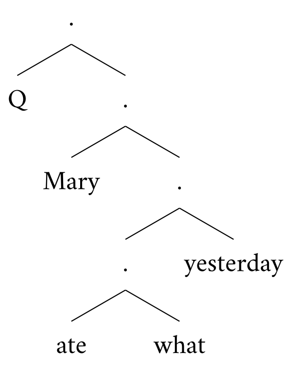
c. what Mary ate yesterday

Let's assume that declarative sentences begin with a declarative complementizer C, and interrogative sentences with an interrogative complementizer Q. In English, the Q complementizer attracts the embedded 'wh' word, but in Whenglish, it does not, as depicted in the trees below.

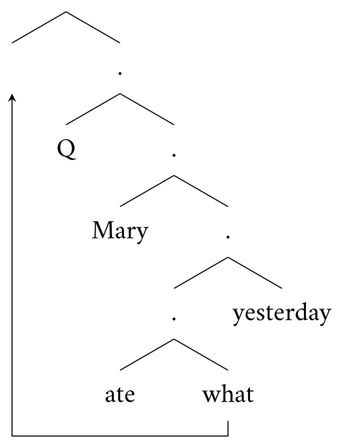
(wh)english



whenglish



english



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.[1] The Q complementizer must co-occur with a wh-word, and vice-versa, as shown in (3).

(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

From this point of view, the *dependency* between a wh-word and a Q complementizer is 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 in Whenglish:

- 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.
- 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`.
- Branching nodes should have exactly two daughters and the label ".".

5. Define a (bottom-up) Finite Tree Automaton `wh1FTA` that enforces these rules. You can choose whatever you like as the state type of the machine. You might like to define a custom type for this purpose, like we did for `polarityFTA`. (If you do, make sure you include `deriving (Eq,Show)`.)

[1] Or perhaps we should say it's pronounced as ε.

```
-- choose a type for q and uncomment this type declaration:
-- wh1FTA :: FTA q
wh1FTA = undefined
```

```
ghci> generates wh1FTA tree_1a
True
ghci> generates wh1FTA tree_1b
True
ghci> generates wh1FTA tree_3a
False
ghci> generates wh1FTA tree_3b
False
ghci> generates wh1FTA (mrg (lf "Q") (lf "what"))
True
ghci> generates wh1FTA (mrg (lf "Q") (lf "Mary"))
False
ghci> generates wh1FTA (mrg (lf "C") (lf "what"))
False
ghci> generates wh1FTA (mrg (lf "Q") (mrg (lf "what") (lf "why")))
True
ghci> generates wh1FTA (mrg (lf "C") (mrg (lf "what") (lf "why")))
False
ghci> generates wh1FTA (mrg (lf "Q") (mrg (lf "John") (lf "what")))
True
ghci> generates wh1FTA (mrg (mrg (lf "Q") (lf "John")) (lf "what"))
False
```

2.2 Island effects

You've probably also learned in syntax classes about the way wh-movement out of certain kinds of structures, called **islands**, is disallowed. Here we'll concentrate on **adjunct islands**.

Adjuncts include relative clauses, as bracketed (4a), and 'because'-clauses, as in (4b).

(4) a. Mary likes the person [that bought books as a gift]

b. Mary laughed [because John bought books as a gift]

As a result, sentences that attempt to move a wh-word out of these constituents are ungrammatical[2]

(5) a. *What does Mary like the person that bought ____as a gift for John?

b. *Why does Mary like the person that bought books ____?

(6) a. *What did John laugh [because Mary bought ____as a gift]?

b. *Why did John laugh [because Mary bought books ____]?

Interestingly, the configurations that block wh-movement in these English examples can also disrupt the dependency between Q and in-situ wh-words. Here's some relevant data from Mandarin Chinese, showing adjuncts in brackets.

2 The interrogatives in (5b) and (6b) are trying to get at questions that would be answered by "as a gift", where the 'why' asks for the reason that the books were bought, not the reason for the liking/laughing.

(7)

*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?’

(8)

*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.

(9)

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

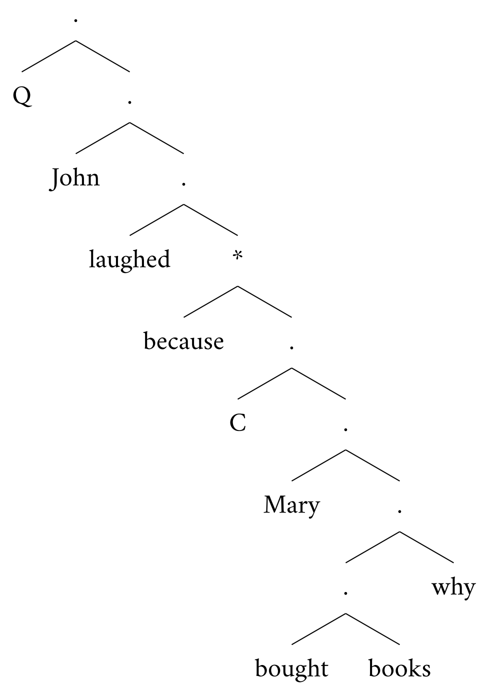
Why did Mary meet the man that gave books to John?

So we’ll assume that Whenglish follows the same pattern.

6. Define a (bottom-up) Finite Tree Automaton `wh2FTA` that enforces the same basic dependency pattern between Q and wh-words as in `wh1FTA`, and additionally guarantees that the licensing Q for a wh-word is not separated from it by an adjunct island boundary. You will probably find that you can use the same states you used for `wh1FTA` here, though you don’t have to. We will use `"*"` as a non-leaf symbol to identify (the root nodes of) adjuncts. Specifically:

   - All of the rules from `wh1FTA` above still apply, *except that* branching nodes can now be labeled either `"."` or `"*"`.
   - 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. And still, every Q complementizer must license at least one wh-word in this way.

   Here’s a tree that should **not** be generated by `wh2FTA`, for example. It is defined for you in `FTA.hs` as `tree_13`.



```
-- choose a type for q and uncomment this type declaration:
-- wh2FTA :: FTA q
wh2FTA = undefined
```

```
ghci> generates wh2FTA tree_13
False
ghci> generates wh2FTA (Node "." [Node "Q" [], Node "*" [Node "C" [], Node "why" []]])
False
ghci> generates wh2FTA (Node "." [Node "Q" [], Node "." [Node "C" [], Node "why" []]])
True
ghci> generates wh2FTA (Node "." [Node "Q" [], Node "*" [Node "C" [], Node "why" []]])
False
ghci> generates wh2FTA (Node "." [Node "C" [], Node "*" [Node "Q" [], Node "why" []]])
True
```