Ling 185A: Assignment 6

Dylan Bumford, Winter 2024

due: Mon, Feb 26

1 Defining CFGs

(1) (a) A simple CFG generating the language $\{a^n b^n \mid n \geq 1\}$ is the following:

- `X -> ab`
- `X -> aXb`

Thinking bottom-up/inside-out, this grammar starts with `ab`, and then successively wraps `a_b` around it.

A closely related grammar gives the language of **(nonempty) even-length palindromes**, $\{ww^R \mid w \in \{\mathtt{d}, \mathtt{p}\}^+\}$, where w^R is w reversed. This language includes any string whose two halves mirror each other, e.g., `dd`, `pp`, `ddppdd`, `dpdppdpd`, etc.

Define a CFG that generates this language. Think about modifying the center-embedding grammar above, and how you'd generate these strings bottom-up/inside-out. You will need four rules. Specify the rules in the comments below:

```
-- Rule 1: undefined
-- Rule 2: undefined
-- Rule 3: undefined
-- Rule 4: undefined
```

(b) Now implement your grammar in Haskell using the `CFG` type from `CFG.hs`. You'll need to define a data type enumerating your nonterminals (see `Cats` and `eng` in `CFG.hs` for an example), and you'll need to convert your grammar into Chomsky Normal Form. The `cfg-intro.pdf` slides do this for the $\mathtt{a}^n\mathtt{b}^n$ grammar, and the methods used for its two rules apply straightforwardly to the four rules you gave above. Keep in mind that the terminal items for `Rule`s are strings (not characters), so your grammar should bottom out with `"d"` and `"p"`, not `'d'` and `'p'`.

```
-- uncomment these next lines and define a type for your nonterminals:
-- data NT = ...
-- palGrammar :: CFG NT
palGrammar = undefined
```

Check that your grammar works by trying to parse some phrases (lists of strings). Make sure some of them are proper palindromes and some aren't. The result of parsing a proper palindrome should be whatever your "top-level" category is. The result of parsing a non-palindrome should be an empty list. So you should get something like:

```
ghci> parse palGrammar (map return "dpdppdpd")
[S]
ghci> parse palGrammar (map return "dpddpdpd")
[]
```

(We have to `map return` over this string to convert it from a list of `Char`s to a list of `String`s, since again in this toy language our "words" are just one character long. Remember: `return = \x -> [x]`)

(2) For every FSA there is a CFG that characterizes exactly the same language. Prove this by writing a function that can convert any FSA to an equivalent CFG. You may want to look at the example conversion demonstrated in the class slides from this week for inspiration.

To do this you'll need the freedom to create nonterminal categories based on the states and symbols of the FSA. Since there can be arbitrarily many of these, depending on the FSA you're handed, use `Int` as your set of nonterminals (the nice thing about integers is that you can never run out of them).

```
fsaToCFG :: FSA -> CFG Int
fsaToCFG (FSAWith states symbols starts stops delta) = undefined
```

2 Parsing

(3) (a) `CFG.hs` defines a CFG `eng`. Among other things, this grammar includes pre-posed 'while'-phrases (like 'while the baby cried, …'), a verb `"watched"` that is ambiguous between transitive and intransitive senses, and categories for relative clauses (both subject-gapped like 'baby that _ cried' and object-gapped like 'baby I watched _') which function as NP modifiers.

This grammar generates center-embedded structures like 'this book every student I know read stinks'. Use `parseLBT` from the `CFG.hs` module to generate a parse tree for this sentence.

```
s1 :: String
s1 = "this book every student I know read stinks"

s1Parsed :: [LBT Cats]
s1Parsed = undefined
```

Once you've successfully defined `s1Parsed`, call `displayForest s1Parsed` to view a graphical representation of the resulting tree structure. Notice in particular the *nested* or *center-embedded* nature of this structure.

(4) (a) The sentence 'while Sam watched the baby that awoke cried' is a **garden-path sentence**. On the first pass through, people tend to misunderstand it, perceiving 'the baby' as the start of a DP that's functioning as the object of 'watched'. A similar effect, albeit a bit less striking, is observed in the simpler sentence 'while Sam watched the baby cried'.

As in the previous question, use `parseLBT` to generate parse trees conforming to the rules of `eng` for 'while Sam watched the baby' and 'while Sam watched the baby cried'.

```
phrase1 :: String
phrase1 = "while Sam watched the baby"

phrase1Parsed :: [LBT Cats]
phrase1Parsed = undefined

phrase2 :: String
phrase2 = "while Sam watched the baby cried"

phrase2Parsed :: [LBT Cats]
phrase2Parsed = undefined
```

Use `displayForest phrase1Parsed` and `displayForest phrase2Parsed` as before to view your trees. Notice the different ways in which the terminals in these trees share are grouped (notice also how a lot turns on the category ambiguity of `"watched"`). In one or two sentences, discuss how these different groupings shed light on how the garden-path effect arises in 'while Sam watched the baby cried'.

```
-- put your thoughts
-- in the comments
-- here
```

(b) Fill in the empty CKY chart below for 'while Sam watched the baby cried'. Each cell in the chart corresponds to a substring of the sentence, and is designated by two `Int`s and a list of categories that that substring can be assigned according to the grammar `eng`. Complete the chart by filling in these lists of categories, where possible. (Some cells should have more than one category, and some should have none.) You should do this by hand, following the

CKY algorithm and seeing how the previous cells you've filled in, together with the production rules of `eng`, are all you need to know to fill in the current cell.

```
chart :: [((Int,Int),[Cats])]
chart = [((0,1),[]),((0,2),[]),((0,3),[]),((0,4),[]),((0,5),[]),((0,6),[]),
                    ((1,2),[]),((1,3),[]),((1,4),[]),((1,5),[]),((1,6),[]),
                               ((2,3),[]),((2,4),[]),((2,5),[]),((2,6),[]),
                                          ((3,4),[]),((3,5),[]),((3,6),[]),
                                                     ((4,5),[]),((4,6),[]),
                                                                ((5,6),[])]
```

See if you can make sense of the way the final chart does justice to the garden-path effect, in that it reflects how the mis-analysis can arise in the first place. Your answers to the first half of the question will help you fill in the chart.

(5) Write a function `parseMT` that parses phrases to trees and weights at the same time. You can do this however you want, though you may wish to define some auxiliary functions analogous to those for the various `parse`ing functions in `CFG.hs`.

```
parseMT :: (Eq nt, Monoid v) => [(Rule nt, v)] -> [String] -> [(LBT nt, v)]
parseMT mcfg [w] = undefined
parseMT mcfg ws  = undefined
```