Assignment 9 (Due Mon Dec 13 at 5pm) Computational Linguistics (LING 455)

(1) A simple CFG generating the language $\{a^n b^n \mid n \geqslant 1\}$ 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**, $\{ww^R \mid w \in \{\mathsf{a},\mathsf{b}\}^+\}$ (note on notation: Σ^+ is Σ^* minus the empty string). This language includes `aa`, `bb`, `aabbaa`, `ababbaba`, etc.

Give a CFG generating this language. Think in terms of **modifying** the grammar above, and how you'd generate these strings bottom-up/inside-out. You will need 4 rules (do not convert the grammar to CNF yet).

- Rule 1:
- Rule 2:
- Rule 3:
- Rule 4:

(2) Implement your grammar in Haskell. You'll need to convert the grammar into Chomsky Normal Form (CNF). The Week 11 slides do this for the $\mathsf{a}^n\mathsf{b}^n$ grammar, and the methods used for its 2 rules apply straighforwardly to the 4 rules you gave above. (See `eng` in `Utils.hs` for a refresher/model of how to encode a CFG in CNF in Haskell.)

```haskell
data PalCats = Undefined
  deriving (Show, Eq)

palGrammar :: CFG PalCats Char
palGrammar = undefined
```

Check that your grammar works as expected by calling `parse palGrammar "ababbaba"` (which should parse successfully to whatever your "top-level" non-terminal category label is) and `parse palGrammar "abaababa"` (which shouldn't parse successfully to that category).

(3) `Utils.hs` defines a slightly enlarged CFG `eng` with relative clauses (both subject-gapped like *baby that _ cried* and object-gapped like *baby I watched _*) which can function as NP modifiers, preposed *while*-phrases, and a verb `"watched"` which is ambiguous between transitive and intransitive senses.

This grammar generates center-embedded structures like *the book this baby I know wrote stinks*. Use `parseToLBT` to generate an `LBT` parse tree for this sentence.

```haskell
s1 :: String
s1 = "the book this baby I know wrote stinks"

s1Parsed :: [LBT Cat String]
s1Parsed = undefined
```

Once you've successfully defined `s1Parsed`, call `displayForest s1Parsed` to view a lovely graphical representation of the resulting tree structure. Notice in particular the *nested* or *center-embedded* nature of this structure — how the innermost noun and verb are grouped, the next-innermost noun and verb, and so on.

(4) When we parse, whether it's to categories, trees, or some other structure, we parse to **lists** of those things, type `[a]`, rather than a single value of type `a` (whatever `a` happens to be). Why? Answer in a sentence or two.

```
{- Your answer goes here -}
```

(5) The sentence *while Sam watched the baby that awoke cried* is a **garden-path sentence**. On the first pass through, people tend to mis-parse 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 (in my own judgment, at least) is observed in the slightly simpler sentence *while Sam watched the baby cried.*

As in the previous question, use `parseToLBT` 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 Cat String]
phrase1Parsed = undefined

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

phrase2Parsed :: [LBT Cat String]
phrase2Parsed = undefined
```

Use `displayForest phrase1Parsed` and `displayForest phrase2Parsed` as before to view your trees. Notice the different ways in which the terminals 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.*

```
{- Your answer goes here -}
```

Fill in the empty CYK 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 may have more than one category!)

```
chart :: [((Int,Int),[Cat])]
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),[])]
```

Make sure that your final chart does justice to the garden-path effect, in that it reflects how the misanalysis can arise in the first place. Your answers to the first half of the question will help you fill in the chart.

(6) In this problem we will consider the garden-path sentence *while Sam watched the baby cried* from the perspective of **transition-based parsing**. Give a **SHIFT-REDUCE** derivation of this sentence by listing, in order, the stages that result from either an application of SHIFT or an application of REDUCE. (Note that your answers in the previous question, and in particular the graphical trees you generated, should prove extremely helpful here!)

You may check that your answer represents a valid derivation by calling `checkDeriv stepSR derivationSR`.

```
derivationSR :: [Stage Cat String]
derivationSR = [ ([] , ["while","Sam","watched","the","baby","cried"]), -- starting stage
                 ([] , []), -- fill in these 10 intermediate stages
                 ([] , []), -- according to the logic of shift/reduce
                 ([] , []), -- parsing, and the rules of the `eng` grammar
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([S], []) ] -- final stage

type Stepper cat term = CFG cat term -> Stage cat term -> [Stage cat term]

checkDeriv :: Stepper Cat String -> [Stage Cat String] -> Bool
checkDeriv stepper (x:y:xs) = elem y (stepper eng x) && checkDeriv stepper (y:xs)
checkDeriv _ _            = True
```

Now that you have a successful parse, put yourself in the shoes of a processor who at first **mis-parses** the sentence in the way we have been considering. How would that derivation have gone differently? Specifically, at what stage in the derivation above could you have made a wrong turn, and what does that wrong turn consist in? Answer in a sentence or two:

```
{- Your answer goes here -}
```

Now, provide a **top-down** derivation of the garden-path sentence, i.e., one using the rules PREDICT and MATCH. You may check that your answer represents a valid derivation by calling `checkDeriv stepTD derivationTD`.

```
derivationTD :: [Stage Cat String]
derivationTD = [ ([S], ["while","Sam","watched","the","baby","cried"]), -- starting stage
                 ([] , []), -- fill in these 10 intermediate stages
                 ([] , []), -- according to the logic of predict/match
                 ([] , []), -- parsing, and the rules of the `eng` grammar
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []),
                 ([] , []) ] -- final stage
```

As before, consider how a **mis-parse** would have gone instead. At what stage in the derivation could the wrong turn have happened, and what does that wrong turn consist in? What differences (if any) do you see from how the SHIFT-REDUCE wrong turn transpires? Answer in a sentence or two.

```
{- Your answer goes here -}
```