Ling 185A: Assignment 7

Dylan Bumford, Winter 2024

due: Wed, Mar 6

1 Left-corner parsing review

Here's a simple CFG you should use for most of the questions below. Note that we are relaxing the Chomsky Normal Form requirement that nonterminals be rewritten as *exactly two* other nonterminals. Instead, we allow nonterminals on the left-hand side to produce *a list* of nonterminals on the right.

S → NP VP	N → baby, boy, actor, award, boss
S → W S S	NP → Mary, John
NP → NP PS 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	C → that
SRC → C VP	PS → 's
ORC → NP V	W → while

Recall that left-corner parsing can be thought of as a combination of bottom-up and top-down parsing. The stack of a left-corner parser mixes (a) symbols with a bar over them, which represent still-to-come constituents (tree nodes with no children yet), and are analogous to the symbols on the stack in top-down parsing, and (b) symbols without a bar over them, which represent already-recognized constituents (tree nodes with no parent yet), and are analogous to the symbols on the stack in bottom-up parsing. The *left-corner* of a context-free rule is the first symbol on the right hand side.

To illustrate, here again is the example we saw in class. The SHIFT and MATCH rules should be relatively easy to understand. They are similar to the rules of the same names in bottom-up and top-down parsing. Notice that the SHIFT rule, since it does bottom-up work, only manipulates "un-barred" symbols (e.g. D at step 1, V at step 5), and the MATCH rule, since it does top-down work, only manipulates "barred" symbols (e.g. $\overline{\text{N}}$ at step 3).

	Type of transition	Rule used	Configuration
0	—	—	($\bar{\text{S}}$, the baby saw the boy)
1	SHIFT	D → the	(D $\bar{\text{S}}$, baby saw the boy)
2	LC-PREDICT	NP → D N	($\overline{\text{N}}$ NP $\bar{\text{S}}$, baby saw the boy)
3	MATCH	N → baby	(NP $\bar{\text{S}}$, saw the boy)
4	LC-CONNECT	S → NP VP	($\overline{\text{VP}}$, saw the boy)
5	SHIFT	V → saw	(V $\overline{\text{VP}}$, the boy)
6	LC-CONNECT	VP → V NP	($\overline{\text{NP}}$, the boy)
7	SHIFT	D → the	(D $\overline{\text{NP}}$, boy)
8	LC-CONNECT	NP → D N	($\overline{\text{N}}$, boy)
9	MATCH	N → boy	(ϵ, ϵ)

The LC-PREDICT rule is a bit more complicated. It manipulates both plain and barred symbols. The application of LC-PREDICT at step 2, for example, transitions from 'D $\bar{\text{S}}$' to '$\overline{\text{N}}$ NP $\bar{\text{S}}$'. This step is based on the fact that D is the left-corner of the rule 'NP → D N': having recognized (in a bottom-up manner) a D, we predict (in a top-down manner) an N; and if we manage to fulfill that prediction of an N, we will have recognized (in a bottom-up manner) an NP.

The LC-CONNECT rule, like the LC-PREDICT, predicts "the rest" of the right-hand side of a rule on the basis of having already recognized the rule's left-corner. For example, when we've reached 'NP $\bar{\text{S}}$', the application of LC-CONNECT at step 4 is based

on the fact that NP is the left-corner of the rule 'S → NP VP'. The difference here, compared to the situation above with LC-PREDICT, is that we already have a *predicted* S (i.e. the symbol $\bar{\text{S}}$) waiting for us there on the stack. Since we've already found an NP bottom-up, the prediction of an S will be fulfilled if we find a VP in the coming words, so it can be swapped for a predicted VP (i.e. the symbol $\overline{\text{VP}}$).

One catch to look out for is that when LC-PREDICT or LC-CONNECT applies to a unary grammar rule, there are no new barred symbols created, because the only thing on the right-hand side of the rule *is* the left-corner. For example, the last step of parsing 'the baby won', using the unary rule 'VP → V', is one such situation.

	Type of step	Rule used	Configuration
0	—	—	($\bar{\text{S}}$, the baby won)
1	SHIFT	D → the	(D $\bar{\text{S}}$, baby won)
2	LC-PREDICT	NP → D N	($\overline{\text{N}}$ NP $\bar{\text{S}}$, baby won)
3	MATCH	N → baby	(NP $\bar{\text{S}}$, won)
4	LC-CONNECT	S → NP VP	($\overline{\text{VP}}$, won)
5	SHIFT	V → won	(V $\overline{\text{VP}}$, ϵ)
6	LC-CONNECT	VP → V	(ϵ, ϵ)

2 Incremental Parsing

Here are some simple type synonyms encoding the components of the tables above.

```haskell
-- in a configuration, an item in memory is either a nonterminal we've already
-- got (un-barred) or a nonterminal we are predicting (barred)
data Chunk = Got Cat | Pre Cat

-- just for convenience, if you like:
got, pre :: Cat -> Chunk
got c = Got c
pre c = Pre c

-- the configuration's memory is just a list of chunks (nonterminals)
type Stack = [Chunk]
-- the buffer is a list of words
type Buffer = [String]
-- the configuration is a pair of a current stack and a current buffer
type Config = (Stack, Buffer)

-- between the three parsing strategies, we have six kinds of transitions
data Transition
  = Shift | Reduce
  | Match | Predict
  | LCPredict | LCConnect

-- one row in a derivation table identifies a transition, a grammar rule, and
-- a new config that results from applying the transition to the previous config
type ParseStep = (Transition, Rule, Config)
-- a table is just a list of rows
type ParseTable = [ParseStep]
```

You will need to provide your answers in this Haskell file, using the types above where appropriate. BUT you are not actually writing any programs here. And you are highly encouraged to work out the derivations with pencil and paper. This file is only for submitting your final responses (and using the type system to prevent any typos.)

(1) For this exercise, you'll investigate the memory requirements imposed by a left-corner parsing schema by constructing left-corner parse tables for six of the sentences from the class handout. We have (1b) "Mary's baby won" and (1c) "Mary's baby's boss won", illustrating **Left Embedding**. We have (2b) "John met the boy that saw the actor" and (2c) "John met the boy that saw the actor that won the award", illustrating **Right Embedding**. And we have (3b) "the actor the boy met won" and (3c) "the actor the boy the baby saw met won", illustrating **Center Embedding**. The crucial question is whether the memory requirements of the parsing algorithm — as measured by the maximum number of symbols on the stack at any given step — increases between each (b) sentence and the corresponding (c) sentence.

For each (b) sentence, please provide the entire derivation table, identifying the transition, the production rule, and the buffer at each step. For each (c) sentence, you don't need to provide the whole table (though you can if you want), **but you need to provide at least the rows containing the configurations that impose the largest memory load.** So if you want, for the (c) sentences, you only need to include these maximally costly row(s) in the `ParseTable` you define.

a. Complete the parse table for (1b): "Mary 's baby won' "

```
ex1b :: ParseTable
ex1b =
  [ -- starting configuration:     ([pre S]                          , ["Mary","'s","baby","won"])
    (Shift     , NP :- "Mary"    , ([got NP, pre S]                , ["'s","baby","won"]))
  , (LCPredict , NP :> [NP,PS,N] , ([pre PS, pre N, got NP, pre S] , ["'s","baby","won"]))
  -- , fill in
  -- , the rest
  -- , of this
  -- , transition table
  , undefined
  ]
```

b. Complete the parse table for (1c): "Mary 's baby 's boss won":

```
ex1c :: ParseTable
ex1c =
  [ -- starting configuration:   ([pre S]         , ["Mary","'s","baby","'s","boss","won"])
    (Shift     , NP :- "Mary"  , ([got NP, pre S] , ["'s","baby","'s","boss","won"]))
  -- , fill in
  -- , the rest
  -- , of this
  -- , transition table
  , undefined
  ]
```

c. Complete the parse table for (2b): "John met the boy that saw the actor"

```
ex2b :: ParseTable
ex2b =
  [ -- starting configuration: , ([pre S]         , ["John","met","the","boy","that","saw","the","act
    (Shift     , NP :- "John"  , ([got NP, pre S] , ["met","the","boy","that","saw","the","actor"]))
  -- , fill in
  -- , the rest
  -- , of this
  -- , transition table
  , undefined
  ]
```

d. Complete the parse table for (2c): "John met the boy that saw the actor that won the award"

```
ex2c :: ParseTable
ex2c =
  [ -- starting configuration: , ([pre S]          , ["John","met","the","boy","that","saw","the","act
    (Shift     , NP :- "John"   , ([got NP, pre S] , ["met","the","boy","that","saw","the","actor","th
  -- , fill in
  -- , the rest
  -- , of this
  -- , transition table
  , undefined
  ]
```

e. Complete the parse table for (3b): "the actor the boy met won"

```
ex3b :: ParseTable
ex3b =
  [ -- starting configuration: , ([pre S]         , ["the","actor","the","boy","met","won"])
    (Shift     , D :- "the"     , ([got D, pre S] , ["actor","the","boy","met","won"]))
  -- , fill in
  -- , the rest
  -- , of this
  -- , transition table
  , undefined
  ]
```

f. Complete the parse table for (3c): "the actor the boy the baby saw met won"

```
ex3c :: ParseTable
ex3c =
  [ -- starting configuration: , ([pre S]         , ["the","actor","the","boy","the","baby","saw","met
    (Shift     , D :- "the"     , ([got D, pre S] , ["actor","the","boy","the","baby","saw","met","won
  -- , fill in
  -- , the rest
  -- , of this
  -- , transition table
  , undefined
  ]
```

g. Based on your tables, as the depth of **Left Embedding** increases, does the memory load required by a left-corner parser increase, decrease, or stay constant?

```
data Monotonicity = Increase | Decrease | Constant

leftEmbeddingMemory :: Monotonicity
leftEmbeddingMemory = undefined
```

As the depth of **Right Embedding** increases, does the memory load required by a left-corner parser increase, decrease, or stay constant?

```
rightEmbeddingMemory :: Monotonicity
rightEmbeddingMemory = undefined
```

As the depth of **Center Embedding** increases, does the memory load required by a left-corner parser increase, decrease, or stay constant?

```
centerEmbeddingMemory :: Monotonicity
centerEmbeddingMemory = undefined
```

(2) Suppose you run into Martian who, oddly, appears to have the context-free grammar defined in Section 1. Then you discover certain new kinds of sentences that this Martian judges to be acceptable sentences of their language, including:

i. John said loudly Mary won
ii. Mary said quietly John 's boss won
iii. 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.

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

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*.

Suppose we know, however, that this Martian uses **bottom-up** parsing. And it's generally accepted that a memory limitation is responsible for the fact that, although this Martians finds iv.@ to be a perfectly unremarkable sentence, v.@ makes their head spin.

So although v., like iv., is generated by the rules in our current grammar and is therefore assumed to in fact be *grammatical* for these Martians, there is something else going wrong when they try to process v.

iv. ✔️ John met the boy
v. 😱 John met the boy that saw the actor

In order to decide between **Hypothesis 1** and **Hypothesis 2**, you ask the Martian for their judgments of the following sentences. The first three are processed without issue, but ix.@ takes the Martian forever to make any sense of.

vi. ✔️ John won
vii. ✔️ Mary said quietly John won
viii. ✔️ John said slowly Mary said quietly John won
ix. 😱 John said slowly John said loudly Mary said quietly John won

How can we use this information to choose between **Hypothesis 1** and **Hypothesis 2**? Explain your reasoning. Notice that in this question, unlike above, 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.

```
{-------------------------------

Write your answer here, inside
this block comment

-------------------------------}
```

(3) In this problem we will consider the garden-path sentence *while John watched the baby cried* from the perspective of **incremental parsing**.

a. Give a **bottom-up** derivation of this sentence by listing, in order, the configurations that result from either an application of SHIFT or an application of REDUCE. (Here, you only need to supply the configuration component of each row, not the transition name or production rule involved.) Remember, all the stack symbols in a bottom up derivation are chunks you've *already seen*, so they should all be un-barred.

```
gardenTableBU :: [Config]
gardenTableBU =
  [ ([], ["while","John","watched","the","baby","cried"]) -- starting config
  -- , fill in
  -- , the rest
  -- , of the steps
  , undefined
  ]
```

Now that you have a successful parse, put yourself in the shoes of a person who at first **mis-parses** the sentence in the way we have been considering. How would that derivation have gone differently? Specifically, at what step 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 here
------------------------------}
```

b. Now, provide a **top-down** derivation of the garden-path sentence using the rules PREDICT and MATCH. Keep in mind, all the stack symbols here are *predictions*, so they should all be barred.

```
gardenTableTD :: [Config]
gardenTableTD =
  [ ([pre S], ["while","John","watched","the","baby","cried"]) -- starting config
  -- , fill in
  -- , the rest
  -- , of the steps
  , undefined
  ]
```

As before, consider how a **mis-parse** would have gone instead. At what step 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 here
------------------------------}
```

c. Let's suppose that the human parsing system "backtracks" when it reaches a dead end configuration and realizes that it took a wrong turn, and has to go back a few steps to start off down a different path. This works simply by looking back (right-to-left) through the buffer in an obvious way (i.e. going back to the point in the sentence where a wrong turn was taken, and trying again). And suppose that this backtracking can be detected in experiments where we track the movements of people's eyes as they read. Explain how we could use our garden path sentence in such an experiment to decide whether humans use a bottom-up parser or a top-down parser. (Assume for this question that these are the only two possibilities.)

```
{------------------------------
Answer in a sentence or two here
------------------------------}
```