CS224N/Ling 237 Homework #1

Due Monday, 14 April 2003, 5pm

A.

1. Provide a phrase structure tree representation for these sentences, and briefly justify or discuss controversial points. Particularly important is that you should place nodes over the groups of words that are constituents, but you should also give suitable labels for those constituents, as discussed in M&S, chapter 3, or in section. (However, there isn't a unique right answer: we'll accept things that are reasonable.)

   (a) Residents were asked to leave their mailboxes open and report any suspicious items to the police.

   (b) Mr. Helder apparently drove his father's 1992 light gray Honda Accord some 3,000 miles across the country since Friday.

2. PP attachment ambiguity question. You are given the following grammar.

```
NP -> Det N
NP --> NP PP
PP --> P NP
N --> {sandwich, pickle, boy, table, condiment, side}
Det --> {the, a}
P --> {with, for, on, as}
```

   (a) How many parses do the following phrases have? You should probably do the first couple yourself; for the later ones, you'll probably want to look at C.2, below to find the answer automatically.

      i. the sandwich on the table
      ii. the sandwich on the table with a pickle
      iii. the sandwich on the table with a pickle as a condiment
      iv. the sandwich on the table with a pickle as a condiment on the side

   The sequence of numbers that one gets in this way are the Catalan numbers. You can find out more about them on the web.[1]

   (b) Suppose a top-down parser tried to parse the phrase:

      the sandwich on the table with a pickle as a condiment

[1] Oops! That's another way to learn the answer to this question.

with the above grammar. Where more than one rule appears for an expansion, our parser attempts them in top-to-bottom order as they are listed in the grammar. Assume that the parser can predict or match found words perfectly (that is, do not consider parses generating different words from those in the string). Our top-down grammar would attempt parses for this NP that do not consume all of the input before arriving at a complete parse. Draw the first three partial parses that the parser would consider.

(c) Which complete parse would our parser come up with first?

B.

Let's use a very simple grammar (where S is the start symbol, and terminals are shown in italic)

S	→	NP VP	N	→	*cats*
VP	→	V	N	→	*claws*
VP	→	V NP	N	→	*people*
NP	→	NP RelCl	Comp	→	*that*
NP	→	N	V	→	*scratch*
RelCl	→	Comp V	V	→	*bite*

Simple (non-tabular) parsing strategies, or in particular, recognizers,[2] can be implemented using a state representation where there is a list (stack or queue) for elements being processed and another for the input sentence. We will assume that sort of representation for the questions here, so a state can be represented on one line. Since drawing search trees as trees can get rather difficult with such representations, we can use two devices:

- We can draw the tree as an indented list, where the indentation of the lines shows the depth of the tree. (This is an especially natural way to think of things when used with depth-first search, since then the order of lines down the page corresponds to the order in which things are done depth first, but it can equally represent any other navigation of a search tree: it's just a tree on its side.)

- We can assume an 'oracle' which at any point chooses only the move (or moves) that leads to a successful parse of the sentence. This allows us to abbreviate things by leaving out all the dead ends. If a sentence only has one parse, or we are only showing one parse, we can also omit the indentation. This corresponds to showing the succession of states down one branch of the search tree – we're ignoring search control, and just showing the moves that we're interested in.

Important notation: We use uppercase latin for nonterminals, lowercase latin for terminals, and greek letters for possibly empty sequences of categories (terminal or nonterminal) – except for α which is either a single category or the empty string, and X which can be either a nonterminal or a terminal. We use a bar over categories or sequences of categories that are predicted versus categories that have been found.

[2]That is, devices that simply determine whether a list of words is in the language defined by the grammar, but don't provide parse trees.

Here is a method for LR (bottom-up shift-reduce) parsing, where we want to parse a string, given in the *input* as a particular category. Note that it assumes that the right hand end of a list of symbols counts as the top of the stack. One starts with the initialization in Begin, and then one can do any of the other moves any number of times in any order. Parsing is successful if the sequence of moves ends with a state that fullfills the Halt criterion. Combining these moves with a search algorithm would give an LR parser.

Begin Place the predicted start symbol on top of the stack (i.e., with a bar over it)

- (Shift) Put the next input symbol on top of the stack
- (Reduce) If γ is on top of the stack and $A \to \gamma$, replace γ by A.
- (Reduce attach) If $\overline{A}\gamma$ is on top of the stack and $A \to \gamma$, remove $\overline{A}\gamma$

Halt Halt with success if the stack is empty and there is no more input

For instance, here is a successful parse of *cats scratch people*:

Stack	Input	Operation
$\overline{S}$	*cats scratch people*	Begin
$\overline{S}$ *cats*	*scratch people*	Shift
$\overline{S}$ N	*scratch people*	Reduce
$\overline{S}$ NP	*scratch people*	Reduce
$\overline{S}$ NP *scratch*	*people*	Shift
$\overline{S}$ NP V	*people*	Reduce
$\overline{S}$ NP V *people*		Shift
$\overline{S}$ NP V N		Reduce
$\overline{S}$ NP V NP		Reduce
$\overline{S}$ NP VP		Reduce
		Reduce attach
		Halt

1. [Warm-up question!] Suppose we didn't have a start symbol, we just wanted to know if there is *some* category from which the whole string can be generated. This makes some linguistic sense. Sometimes a 'sentence' in a newspaper (or the Stanford Bulletin) is just a noun phrase, and people commonly respond to questions (like *When will you get around to the asssignment?* with fragments such as PPs (*During next week.*). How would one modify [hint: simplify!] the above parser specification for this case?

2. Top-down parsing.

   (a) Give a similar specific set of operations that implement a top-down, left-to-right (LL) parser. (Hint: you will want to predict a lot more categories than in the above example!)

   (b) Trace the successful parse move sequence for the sentence *cats scratch people* using the grammar above. Assume that we are literally doing top-down parsing right to the level of guessing words.

   (c) Suppose we added one or more rules that rewrote things as empty to the grammar. (In particular, you might consider the rule NP → e, and parsing the sentence *cats bite.*) Does your top-down parser need modification to handle this case correctly? If so, how? If not, say briefly why not?

3. We noted that top-down parsers have a problem with left-recursive rules. For example, they will have problems with the NP → NP RelCl rule in the grammar above. Joe Genius notes that this problem has a ready solution for this grammar. He rejigs his top-down parser to expand rules from right-to-left, rather than left-to-right.

   (a) Does this fix the problem for this grammar? Show the search tree of a top-down recognizer on the sentence *cats scratch people that bite*. This time, we do want the whole search tree, not just the successful path, but you can use the indented list notation. However, to make things less painful, assume that the input is represented as the part of speech sequence N V N Comp V, rather than the actual words, so as to avoid the final top-down prediction of terminals.

   (b) Since Joe is a genius, he knows that this isn't a full solution to the problem, because his new parser will have problems with right recursive rules like S → AdvP S. (For instance, we might suggest such a rule to parse *[Most of the time] I go to class.*) However, he comes up with the following ingenious idea. For each grammar rule, if it is left recursive, he will expand its categories from right-to-left, otherwise, he will expand it from left-to-right, as previously. (Carrying this idea through necessitates a number of further complexities involving maintaining a stack of parts of the sentence that one has parsed, but we'll leave the details to Joe to work out.) Is this a full solution to the problem of edge-recursion? If not, what kinds of grammar rules will still cause problems for Joe's parser? Give a realistic example of a place in English grammar where this problem turns up.

   (c) We presented the problem with top-down parsing in terms of a problem handling left-recursive rules of the form $X \rightarrow X\ \gamma$. But the problem is actually a bit more general than that. What is the more general statement of when a top-down parser runs into trouble?

4. Here are the corresponding moves for a left-corner parser. This time we have to regard the left end of any lists as the top of the stack. (It's common to need to change this convention, so that we can use grammar rules to match sequences in different orders without having to reverse them, which makes them hard to read.)

   Begin Place the predicted start symbol on the stack.

   - (Shift) Put the next input symbol on top of the stack
   - (Pop) If $\overline{a}$ is on top of the stack, and a is the next input symbol, remove both.
   - (Leftcorner attach) If $X\overline{A}$ is on top of the stack and $A \rightarrow X\gamma$, replace $X\overline{A}$ by $\overline{\gamma}$
   - (Leftcorner) If X is on top of the stack and $A \rightarrow X\gamma$, replace X by $\overline{\gamma}A$

   Halt Halt with success if the stack is empty and there is no more input

   The idea of a left-corner parser is it mixes top-down and bottom-up processing: it both works predictively down from a goal, and up from an identified left-corner of a phrase.

   (a) Trace the moves of one successful parse of the sentence *cats scratch people that bite* using the grammar at the beginning.

(b) Through the "Leftcorner attach" operation, this parser does what is sometimes referred to as "composition". Namely, if it has found a Det, and the goal is an $\overline{NP}$, and there is a rule NP → Det Adj N, then it will in one step remove the Det and the $\overline{NP}$ goal, and put goals of an $\overline{Adj}$ and $\overline{N}$ on the stack. A completed NP never actually appears on the stack. Change the above parser so that it doesn't do composition. That is, the "Leftcorner attach" operation would be removed, and the completed NP would be placed on the stack using the existing Leftcorner operation. Modify the parser as needed so as to still have a sound and complete parser.

5. All of the methods we have looked at above can be regarded as instances of "Generalized Left Corner Parsing" (A. J. Demers, Generalized left corner parsing, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pp. 170–181, 1977). Here's the algorithm:

Begin Place the predicted start symbol on the stack (i.e., with a bar over it).

- (Shift) Put the next input symbol on top of the stack
- (LeftPart) If β is on top of the stack and condition Φ holds and $A \to \beta\gamma$, replace β by $\overline{\gamma}A$
- (Attach) If $X\overline{X}$ is on top of the stack, remove both.

Halt Halt with success if the stack is empty and there is no more input

All of top-down (LL), bottom-up (LR), and left-corner (LC) parsing can be realized by the above algorithm, by appropriately defining the condition Φ (which might depend on β, γ, etc.). Define what Φ should be to give each of these strategies.

C. (Practical)

1. One way to explore the Penn treebank is with the program `tgrep`. This only runs on SUN machines (elaine, myth, epic, saga). It requires the following setup:

```
setenv TGREP_CORPUS /afs/ir/data/linguistic-data/Treebank/tgrepabl/wsj_mrg.crp
setenv PATH /afs/ir/data/linguistic-data/bin/sun4x_57:$PATH
setenv MANPATH /afs/ir/data/linguistic-data/man:$MANPATH
```

or if you don't have a defined MANPATH, just:

```
setenv MANPATH /afs/ir/data/linguistic-data/man:
```

You can then give queries like `tgrep 'VP < (NP . NP)'` to find ditransitive verb phrases in the Penn treebank. See `man tgrepdoc` for information on the syntax of tgrep patterns. Patterns almost always have to be quoted so that they are not misinterpreted by the shell.

(a) Use a tgrep query to find three adjectives that take an SBAR complement in the WSJ Penn Treebank (the basic tag for adjectives is JJ, and complex adjective constructions appear in adjectival phrases labeled ADJP. Give the query and the adjectives (there are more than 3 in total; but listing 3 is sufficient).

(b) Which adjective is the (unique) one that occurs commonly with an SBAR complements in the Penn treebank?

(c) In English prepositional phrases normally follow the noun phrase object, giving a structure like:

[[NP She] [VP opened [NP the door] [PP with a crowbar]]]

whereas the reverse sounds somewhat weird:

[[NP She] [VP opened [PP with a crowbar] [NP the door]]]

However, in many circumstances, this ordering can be reversed. This is referred to in the linguistics literature as *Heavy NP Shift* (since one of the reasons for it is the 'heaviness' of the NP, though there are also other causes such as the informational content of the sentence). Find an instance of Heavy NP Shift in the Penn Treebank. Give the instance, and the tgrep query you used to find it.

2. We've put in /afs/ir/class/cs224n/parser a context-free grammar chart parser. It was actually written for probabilistic parsing and so has a couple of hold-overs from that (which I should really clean up!): rules and lexical entries have to have scores specified, even if one is just using it as a nonprobabilistic parser, and it doesn't handle empty nodes. But we can live with these limitations. You can invoke it as follows:

```
cd /afs/ir/class/cs224n/parser
./parse simple-test.grammar simple-test.lexicon s simple-test.sentences
```

The parse file is just a simple shell script that invokes a Java parser. For files in your own directory you can invoke it as:

/afs/ir/class/cs224n/parser/parse *arguments*

The parser takes four arguments: a grammar file, a lexicon file, the start category for parsing, and a file of sentences to parse, one per line. The format of the grammar and lexicon is:

```
# lines beginning with a hash are ignored
cat -> cat+ %% score
# A rule may not have an empty righthand side.
# It must have a score, even if this is going to be
# ignored later.  This can be any number -- e.g., always 1.0
```

Write a grammar and lexicon that can parse the two sentences and all the noun phrases in part A with this parser, and include the grammar and lexicon in your homework. The grammar and lexicon only have to cover the words and constructions in this sentence, but they should use "sensible" rules reflecting reasonably the nature of English grammar. E.g., you should *not* submit a finite state grammar that generates those words. (Given the limitation mentioned above, the grammar can't have empty categories, and so you need to remove those from any rules where one might be tempted to postulate them.)