CS224N/Ling 237 Homework #2
Written Assignment: Parsing Algorithms

Due: Wed, April 14 2004

This portion of the homework should be done individually See the statement on homework collaboration policy on the website `http://cs224n.stanford.edu`

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

S	$\longrightarrow$	NP VP	N	$\longrightarrow$	*cats*
VP	$\longrightarrow$	V	N	$\longrightarrow$	*claws*
VP	$\longrightarrow$	V NP	N	$\longrightarrow$	*people*
NP	$\longrightarrow$	NP RelCl	Comp	$\longrightarrow$	*that*
NP	$\longrightarrow$	N	V	$\longrightarrow$	*scratch*
RelCl	$\longrightarrow$	Comp V	V	$\longrightarrow$	*bite*

Simple (non-tabular) parsing strategies, or in particular, recognizers[1], 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 can get rather difficult with such representation, we can use two devices:

- We can draw the trees 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

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

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.

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 fulfills 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 $A \longrightarrow \gamma$, replace γ with A
- (Reduce attach) If $\overline{A}\gamma$ is on top of the stack and $A \longrightarrow \gamma$, remove $\overline{A}\gamma$.

Halt Halt with a 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 assignment?*) 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 $\longrightarrow$ e, and parsing the sentence *cats bite*.) Does your top-down parser need moficiation 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-recursion rules. For example, they will have problems with the NP $\longrightarrow$ 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. Think about how this would fix the problem.

   (a) 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 $\longrightarrow$ 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.

   (b) We presented the problem with top-down parsing in terms of a problem handling left-recursion rules of the form $X \longrightarrow X\ \gamma$. 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 $\overline{\alpha}$ is on top of the stack, and α is the next input symbol, remove both.
- (Leftcorner attach) If $X\overline{A}$ is on top of the stack and $A \longrightarrow X\gamma$, replace $X\overline{A}$ by $\overline{\gamma}$.
- (Leftcorner) If X is on top of the stack and $A \longrightarrow X\gamma$, 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 Set, and the goal is an $\overline{NP}$, and there is a rule NP $\longrightarrow$ 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 "Leftmost attach" operation would be removed, and the completed NP would be placed on the stack using the exisiting 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 $A \longrightarrow \beta\gamma$, 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 the 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.