CS164: Written Assignment 3
(On Parsing)

Assigned: Thursday, Sep 30, 2004
Due: This assignment is not going to be collected or graded.
We are going to post the solutions on Sunday, Oct 3.

Note: Tuesday, Oct 5, is the day of the first Midterm Exam. Solving the problems in this assignment could prove useful on the midterm exam.

1 Recursive Descent Parsers

Despite its shortcomings, a recursive descent parser is a perfectly fine alternative when what you need to parse is a simple language: such a parser can be written by hand in a straightforward way (see the lecture slides), and it performs well when built from a suitable grammar. To help you learn how to write such parsers, this problem will make you familiar with intricacies of recursive descent parsers. Specifically, the problem will show that sometimes you may want to do more with the grammar than just eliminate its left recursion.

1.1 The Order of Productions

First, unlike other parsers we studied, a recursive descent parser is sensitive to the order of productions. That is, a recursive descent parser *may* behave differently when the productions of a non-terminal A are rewritten from $A \rightarrow B \mid C$ to the form $A \rightarrow C \mid B$.

Question: To see when the order of productions matters, consider the following grammar G. Note that E is the only non-terminal in the grammar. Also note that $id[E]$ denotes an array element.

$$E \rightarrow id \mid id\ [\ E\ ] \mid (\ E\ )$$

1. Construct the recursive descent parser for this grammar. Make sure that your parser evaluates the three productions in the same order as given above. You need to write only four procedures (you can assume that the procedure `term()` defined in the lecture notes is available to you from a library.
2. Give one sequence of tokens that belongs to $L(G)$ (i.e., the language of the grammar) but is not recognized by the parser. That is, give an input on which the parser does not behave as desired.
3. Briefly explain why the parser does not recognize the above sequence of tokens. Your answer must clearly identify the root of the problem.

4. The problem you identified could be fixed by "repairing" the parser, i.e., by making its backtracking more complete. We are not going to do this. Instead, let's modify the grammar slightly. So, give a modified grammar G such that the recursive descent parser recognizes all its sequences of tokens.

5. To see why grammars for recursive descent parsers are actually quite clean, write the LL(1) version of the grammar.

1.2 Performance of Recursive Descent Parsers

The reason why a recursive descent parser is more powerful than a predictive (LL(1)) parser (which means that it can recognize some grammars that the predictive parser cannot) is that recursive descent parsers rely on backtracking. Not surprisingly, this backtracking can be a source of performance problems (i.e., the parser could be too slow).

Question: Consider the following grammar. Note that the grammar is not left recursive, and so it is suitable for the recursive descent parsing. Also, you should know that the grammar does not have a problem like the one described in Question 1 (i.e., a recursive descent parser will recognize all strings from the language of the grammar).

$$\begin{aligned} E &\rightarrow T + E \mid T - E \mid T \\ T &\rightarrow F * E \mid F \,/\, E \mid F \\ F &\rightarrow int \mid (\,E\,) \end{aligned}$$

1. How many calls to `F()` is the parser going to make on the input "3"?

2. How many calls to `F()` is the parser going to make on the input "3*3"?

3. Fix the grammar to make the recursive descent parser highly efficient. Hints: You can use one of the grammar transformations covered in the lectures; make sure your transformation doesn't create a problem like the one discussed in Question 1.

2 LR Parsing

Consider the grammar below. This question is about non-deterministic LR parsing of the string `int + ( int )`.

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow int \mid (\,E\,) \end{aligned}$$

Question:

1. Show the parsing sequence of actions performed by the instance that succeeds in parsign the input string.

2. Show a parsing sequence of actions performed by an instance that fails to parse the input string.

3. Show a stack configuration that is guaranteed to lead to a stuck state. Justify your answer (that is, why are you sure this LR stack is doomed?).