CS143 Handout #12
Autumn 2000

PS2: Shift-reduce parsing

Due 5:00pm Wed Oct 25th
(can be submitted at most **2** days late)

The goal

This week's assignment is an exploration into the techniques and details of bottom-up shift-reduce parsing. It consists of some written problems and thought questions. Your solution should be submitted **on paper**, not electronically. You can hand your assignment in class or put it in the box outside Gates 192. Assignments received before 5pm Wednesday are considered on-time, up til 5pm Thursday is one day late, 5pm Friday is two days late. Assignments will not be accepted after 5pm Friday, in order that we can publish the solution to help you study for the upcoming midterm.

Although you can discuss ideas with others, you are expected to submit your own independent work. Please refer to web site for details of the collaboration policy as it applies to problem sets to be sure you understand the ground rules.

We haven't had much chance to see your handwriting yet, but history shows that folks who type for a living are not usually front-runners in the Best Penmanship category (evidence your instructor's lowercase a's on the board), but the TAs will certainly appreciate your efforts to ensure your solution is tidy and legible.

Where we ask you to briefly explain, let us emphasize the word "briefly". Correct answers to the short answer questions can and should be stated in just a few sentences. Keep it relevant and factual!

1) (20 pts) Consider the following grammar:

```
S    ← AbA
A    ← Aa | a
```

a) Build the family of SLR(1) configurating sets for this grammar in goto-graph form. Each set should be numbered and the successors should be indicated as arrows between sets. Each arrow should be labelled with the appropriate symbol.

b) Trace the operation of an SLR(1) parser on the input aaba. Show the contents of the state stack, the remaining input, and the parser action taken at each step.

2) (25 pts) You are given the following grammar:

```
S    ← xAy | yA | xB
A    ← a | ∂
B    ← b | ∂
```

a) Construct the complete LR(1) parsing table for this grammar. Leave error entries blank.

b) Is this grammar SLR(1)? Briefly explain.

c) Is this grammar LALR(1)? Briefly explain.

3) (15 pts) One of the least loved and least understood aspects of the C programming language is its type declarators. Let’s take a look at it from a parsing perspective to see if we can help unshroud some of the mystery and get more practice with LR table construction. Here is a simplified grammar for C declarations:

```
S     ← TD
T     ← int | char
D     ← id | *D | D[]
```

T is a type name, D is a declarator, which can either be simple or a pointer or array variant. The terminals for this language are the type names (int and char), identifiers, and the various punctuation symbols. This grammar only covers a small subset of valid C declarations— full C has many other type names, type qualifiers, function types, and so on.

To answer the following question, you will need to first build the SLR(1) configurating sets and sketch the parse table. You do not need to hand these in, just do it on scratch paper. (More practice with the LR table construction is always a good thing!)

a) The SLR(1) table for grammar has one or more conflicts that indicate it is not SLR(1). Identify each conflict and give an example input that exhibits that conflict.

b) Do the conflicts arise from limitations of the SLR(1) method or from a fundamental issue that renders the grammar invalid for LR parsing? Asked another way, would using LALR(1) parsing resolve the conflicts? If not LALR(1), what about canonical LR(1)? Briefly explain.

4) (10 pts) One way to resolve conflicts in a LR parsing table is to remove all but one of the conflicting entries, thus forcing a choice. For example, the dangling-else ambiguity can be resolved by ignoring the possible reduce of the inner if-statement and always shifting. This causes the else clause to bind to the inner statement. The yacc parser generator will automatically resolve any shift/reduce conflict in favor of the shift. Provide an intuitive argument for why the shift is preferred. (Hint: if it chose a reduction instead, what effect might that have on the language being recognized?)

5) (10 pts) You will recall that for a grammar to be LL(1), it cannot have any left-recursive productions and thus we had to transform those productions into equivalent right-recursive forms. In LR parsing, neither left nor right recursion is a problem. However, this is not to say that both are handled equally efficiently. Consider the growth of the LR parse stack with left recursion versus right recursion and explain which form is preferable if efficiency is a concern.

If you can read this, you're holding the handout upside-down. No wonder the assignment didn't make sense!