CS143 Midterm
Fall 2009

- Please read all instructions (including these) carefully.
- There are 4 questions on the exam, all with multiple parts. You have 75 minutes to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	20	
2	20	
3	20	
4	15	
TOTAL		

1. **Lexical Analysis** (20 points)

(a) Consider the string:

```
babcaababccbcabb
```

And the following sequence of tokens produced by the lexer:

```
b abc aab abcc b c abb
```

Give a minimal flex specification (i.e. with a minimal number of rules) that produces this tokenization. Each rule should be as simple as possible. You need not give actions for the rules, just the regular expression matching part.
Your rules must adhere to (be in the language of) the following grammar:

$$R \rightarrow a \,|\, b \,|\, c \,|\, R^* \,|\, RR$$

So, for example, cb^*a is permitted, but $a + b$ and $c?b^+.$ are examples of expressions that are not allowed.

```
There are a number of possible 3 rule solutions.  A straightforward one is

c
b
aa*b*c*

Some people managed to find a solution with 2 rules:

aa*b*c*
c*b*

Any correct 2 or 3 rule solution received full credit.
```

(b) Consider the following flex specification that your co-worker has written:

```
c             { printf("1"); }
ac+b*         { printf("2"); }
cc            { printf("3"); }
ab*           { printf("4"); }
a             { printf("5"); }
ac*b+         { printf("6"); }
```

Since you have taken CS143 you immediately spot one or more rules that can never be executed. Which rules can never apply and why?

Rule 5 (a) can never be executed, because it is included in rule 4 (ab*).

Rule 6 (ac*b+) can never be executed, because when there is no c (i.e. ab+) it is included in rule 4 (ab*) and when there are one or more c's (i.e. ac+b+) it is included in rule 2 (ac+b*).

2. **Top-Down Parsing** (20 points)

Consider the following grammar over the alphabet a, b, c:

$$
\begin{aligned}
S &\rightarrow Xa \\
X &\rightarrow bX \\
X &\rightarrow Y \\
Y &\rightarrow Zc \\
Z &\rightarrow bZ \\
Z &\rightarrow \epsilon
\end{aligned}
$$

(a) Prove that this grammar is not LL(1).

The easiest way to show that the grammar is not LL(1) is to show that it is ambiguous. For instance, consider the string "bca"

S->Xa->bXa->bYa->bZca->bca
S->Xa->Ya->Zca->bZca->bca

These two leftmost derivations form two different parse trees, and therefore the grammar is ambiguous.

Some students also constructed an LL(1) parsing table and showed that there is a conflict in this table; this is also a valid answer. Some students argued why this grammar causes ambiguity during an LL(1) parse; for those answers the number of awarded points depends on the clarity and specificity of the explanation.

(b) It is possible to drop exactly one production from this grammar to obtain a new grammar generating the same language that is an LL(1) grammar. Identify this production and prove that the resulting grammar is LL(1).

It is possible to drop either \verb+X -> bX+ or \verb+Z ->bZ+.
Assuming we drop \verb+Z->bZ+, we get the following parsing table:

	a	b	c	$
S		Xa	Xa	
X		bX	Y	
Y			Zc	
Z			\epsilon	

Since there are no conflicts in this table, the new grammar is LL(1). A common mistake in this question was to only argue that the new grammar is unambiguous, left-factored and not left-recursive and

claim that it follows from these claims that the grammar must be LL(1).

3. **Semantic Actions** (20 points)
Consider the following context-free grammar decribing boolean formulas:

$$S \to F$$
$$F \to F \wedge F \,|\, F \vee F \,|\, \neg F \,|\, (F) \,|\, id \,|\, true \,|\, false$$

Here F is the only non-terminal, *id* is a terminal denoting a boolean variable, and *true* and *false* are terminals associated with boolean constants.

(a) Give semantic actions to compute the truth value of a formula. Assume all variables have the value *true*.

We define a synthesized attribute "val" that represents the truth value of each subformula, and compute it using semantic actions. The following solution uses bison notation, but we were happy with any reasonable notation for attributes.

```
S-> F {$$.val = $1.val }

F->  F | F {$$.val = $1.val || $2.val}
   | F & F {$$.val = $1.val && $2.val}
   | !F    {$$.val = !$1.val}
   | (F)   {$$.val = $1.val}
   | id    {$$.val = true}
   | true  {$$.val = true}
   | false {$$.val = false}
```

(b) A literal (i.e., a leaf in the parse tree) is said to have positive phase if it appears under an even number of negation symbols, and has negative phase if it appears under an odd number of negation symbols. (For example, in the formula, $\neg(a \vee \neg(b \wedge \neg c))$, a and c have negative phase, and b has positive phase). Give semantic actions to compute the phase of each literal in a formula.

It is most natural to compute the phase of literals top-down. We define an inherited attribute "phase", and compute it as follows:

```
S -> F {$$.phase = positive; $1.phase = positive}

F->  F | F {$1.phase = $$.phase; $2.phase = $$.phase}
   | F & F {$1.phase = $$.phase; $2.phase = $$.phase}
   | !F    {$1.phase = ($$.phase == positive ? negative : positive) }
   | (F)   {$1.phase = $$.phase}
   | id    {$1.phase = $$.phase}
   | true  {$1.phase = $$.phase}
   | false {$1.phase = $$.phase}
```

Common mistakes:
- Not defining any attributes or not giving semantic actions
- Trying to compute phase bottom-up using synthesized attributes
- Many people tried to use global state to compute phase, but obtained incorrect

4. **Bottom-Up Parsing** (15 points)

(a) Prove that there is a grammar with a single production that has a shift-reduce conflict under SLR(1) parsing rules.

```
S -> SaS

Considering just this single production, the states for the SLR
automaton are:

state 1: { S -> .SaS }
state 2: { S -> S.aS }
state 3: { S -> Sa.S, S -> .SaS }
state 4: { S -> SaS., S -> S.aS }

Some of the transitions are 1 -S-> 2, 2 -a-> 3, and 3 -S-> 4.
Clearly terminal 'a' is in Follow(S), so state 4 has a shift-reduce conflict.
If we include the production S' -> S that is added automatically by a
parser generator the result is the same, just slightly more complicated.

The answer is short enough that it can be discovered by brute force,
trying all single productions with right-hand sides of increasing
length.  However, with a little bit of reasoning we can quickly
eliminate most of the possibilities; the following rather long explanation
also covers the variety of mistakes made on this problem.

First, since we are after a shift-reduce conflict we immediately know two
things:

(1) Since only terminals are involved in shift moves, the right-hand
side of the production must include at least one terminal.  Thus we
can rule out answers such as S -> \epsilon, S -> S, and S -> SS.

(2) All conflicts involve at least two distinct items in some state of
a parsing automaton.  The only way to get more than one item in a
state is to, at some point, have an item of the form S -> \alpha .X
\beta for some non-terminal X, which will pull all items of the form X
-> . \gamma into the state.  Since the answer is a one production
grammar, we also know that X = S.

Putting (1) and (2) together, we know we need a production S ->...
```

with at least one terminal 'a' and one 'S' on the right-hand side. Now,

S -> Sa

won't work. The DFA states are

{ S -> .Sa }
{ S -> S.a }
{ S -> Sa. }

All states have only 1 item in them and it is necessary to have at least 2 items in some state to even have the possibility of a shift-reduce conflict. The other possible two symbol rhs is

S -> aS

which is definitely more promising, as the DFA has states:

(1) { S -> .a S }
(2) { S -> a.S, S -> .aS}
(3) { S -> aS. }

We are trying to get a state like (2), but with a reduce move, which requires the dot on some item be all the way to the right. State (2) moves to reducing state (3) on an 'S' transition, but (3) has only one item because the second item in state (2) starts with 'a', not 'S' and thus does not contribute an item to state (3). But if the production started with 'S' instead of 'a' then the second item in state (2) would have the form S -> .S... and would contribute an item to state (3), and so we might get a shift-reduce conflict. In fact this line of thought leads to the solution S -> SaS

The only other plausible short answer is S -> aSa, but this doesn't work for the same reason that S -> aS doesn't work. Consider the state { S -> a.Sa, S -> .aSa }. On an 'S' transition the resulting state is { S -> aS.a }, which has only one item and, since there are no more non-terminals to the right of the '.', can only lead to states with one item. In fact, for this reason, there is no answer that does not include at least two uses of 'S' on the right-hand side. The solution S -> SaS is the shortest such answer.

As an aside, it is not important that the solution grammar generates the empty language; it is a grammar. The answer to this question shows that even adding a single infix operation to a language (e.g., E -> E+E) results in a shift-reduce conflict in a bottom-up parser solely because of that production, regardless of what else is in the grammar.

(b) Prove that there is no grammar with a single production that has a reduce-reduce conflict under SLR(1) parsing rules.

A reduce-reduce conflict, by definition, involves two distinct productions that can be reduced in the same parser state. A single production can never have a reduce-reduce conflict with itself, and so no grammar with one production can have a reduce-reduce conflict.