CS143 Midterm Exam

This midterm exam is open-book, open-note, open-computer, but closed-network. This means that if you want to have your laptop with you when you take the exam, that's perfectly fine, but you **must not** use a network connection. You should only use your computer to look at notes you've downloaded from online in advance. Please write all your answers to this exam on the exam itself. No electronic submissions will be considered without prior consent of the course staff.

If you are a remote student, you may start this exam in any two-hour period between 11:00 AM Pacific time, July 25, and 11:00 AM Pacific time, July 26. If you have any questions, please feel free to call me at (916) 296-7146 any time in this window except between midnight and 8AM on July 26, Pacific time. You may submit the exam either by faxing it to (650) 723-6092 or by scanning and emailing it to cs143-sum1112-staff@lists.stanford.edu. We will send you a confirmation email once we receive your exam.

SUNetID: ______________________

Last Name: ______________________

First Name: ______________________

I accept both the letter and the spirit of the honor code. I have not received any assistance on this test, nor will I give any.

(signed) ______________________________________

You have two hours to complete this midterm. There are 120 total points, which means that as a rough heuristic you might want to spend about one minute per point. This midterm is worth 20% of your total grade in this course.

Question		**Points**	**Grader**
(1) Scanning	(15)	/ 15	
(2) LL Parsing	(30)	/ 30	
(3) LR Parsing	(50)	/ 50	
(4) Comparative Parsing	(25)	/ 25	
	(120)	**/ 120**	

Good Luck!

Problem One: Scanning (15 Points)

Consider the following **flex** script:

```
%%
aa              { printf("1"); }
b?a+b?          { printf("2"); }
b?a*b?          { printf("3"); }
.|\n            { printf("4"); }
```

a. Give an example of an input to this scanner that will produce **123** as an output, or explain why one does not exist.

b. Give an example of an input to this scanner that will produce **321** as an output, or explain why one does not exist.

Problem Two: LL(1) Parsing (30 Points Total)

This question concerns the following grammar, which generates email addresses:

$$Addr \rightarrow Name\ \texttt{@}\ Name\ \texttt{.}\ \texttt{id}$$

$$Name \rightarrow \texttt{id} \mid \texttt{id}\ \texttt{.}\ Name$$

For example, this could generate the addresses

`id@id.id`

`id.id@id.id.id.id`

`id.id.id@id.id`

(i) LL(1) Conflicts (15 Points)

This grammar, as written, is not LL(1). Rewrite the grammar to eliminate all LL(1) conflicts.

(ii) FIRST and FOLLOW Sets **(5 Points)**

Construct the FIRST and FOLLOW sets for all nonterminals in your rewritten grammar.

(iii) LL(1) Parse Tables **(10 Points)**

Using your results from part (ii), construct the LL(1) parse table for your updated grammar. If you identify any LL(1) conflicts, don't worry; just put all applicable entries in the table. In other words, if you discover now that your grammar is not LL(1), we won't hold that against you for this part of the problem.

To save you time, we have provided columns for the table. You just need to provide the rows.

	`id`	`.`	`@`	`$`

Problem 3: LR Parsing (50 Points Total)

Consider the following (already augmented) grammar, which is known to be LR(1):

$$\begin{aligned} S &\rightarrow Z \\ Z &\rightarrow \mathtt{a}M\mathtt{a} \mid \mathtt{b}M\mathtt{b} \mid \mathtt{a}R\mathtt{b} \mid \mathtt{b}R\mathtt{a} \\ M &\rightarrow \mathtt{c} \\ R &\rightarrow \mathtt{c} \end{aligned}$$

(i) LR(1) Parsing (15 Points)

Draw the states of the LR(1) automaton encountered during a parse of the input **`aca`**. You should not construct the entire LR(1) automaton; instead, just show the states of the automaton that you actually use during the parse. If you accidentally construct irrelevant states, **cross them out**. Do not add or change any productions in the grammar.

(ii) SLR(1) and LALR(1) Parsing (15 Points)

a. Given the grammar and the subset of the LR(1) automaton that you constructed in part (i), can you determine whether this grammar is SLR(1)? If you can decide whether the grammar is SLR(1), do so and explain your reasoning. If you cannot decide, explain why not.

b. Given the grammar and the subset of the LR(1) automaton that you constructed in part (i), can you determine whether this grammar is LALR(1)? If you can decide whether the grammar is LALR(1), do so and explain your reasoning. If you cannot decide, explain why not.

(iii) Parsing Efficiency (20 Points)

Consider the following (already augmented) grammars for the language **a***:

$$S \to A$$
$$A \to A\mathbf{a} \mid \varepsilon$$

and

$$S \to A$$
$$A \to \mathbf{a}A \mid \varepsilon$$

Both of these grammars are SLR(1). However, the SLR(1) parser for one of these grammars will use O(n) space in its parsing stack when run on the string $\mathbf{a}^n$, while the other parser will only use O(1) stack space.

Identify which grammar's parser uses O(n) stack space and which grammar's parser uses O(1) stack space. Justify your answer by making specific references to how an SLR(1) parser for **each** grammar parses strings of the form $\mathbf{a}^n$. In other words, even if you can figure out why one parser uses O(n) stack space, you should still explain why the other parser uses only O(1) stack space.

(Extra space for part (iii), if you need it)

Problem Four: Comparative Parsing (25 Points Total)

(i) LL(1) and LR(1) Parsing (15 Points)

Consider the following (already-augmented) grammar, which is both LL(1) and LR(1):

$S \to A$	(1)
$A \to \mathbf{a}BE$	(2)
$B \to \mathbf{b}CD$	(3)
$C \to \mathbf{c}$	(4)
$D \to \mathbf{d}$	(5)
$E \to \mathbf{e}FG$	(6)
$F \to \mathbf{f}$	(7)
$G \to \mathbf{g}$	(8)

This question has two parts.

a. List which productions are performed and in which order when parsing the string **`abcdefg`** with an **LL(1) parser**. Explain why they are performed in this order.

b. List which reductions are performed and in which order when parsing the string **`abcdefg`** with an **LR(1) parser**. Explain why they are performed in this order.

(ii) Manual Conflict Resolution (10 Points)

When writing LR parsers, it is common to introduce precedence and associativity declarations to allow the parser to parse ambiguous grammars. However, these declarations cannot be used in LL(1) parsers. This question explores why.

Consider the following ambiguous grammar:

$$
\begin{aligned}
E &\to E \,\mathbf{+}\, E \\
E &\to E \,\mathbf{*}\, E \\
E &\to \mathtt{int} \\
E &\to \mathbf{(}E\mathbf{)}
\end{aligned}
$$

Explain why this grammar cannot be parsed with an LL(1) parser, even if the parser knew the relative precedences and associativities of addition and multiplication.