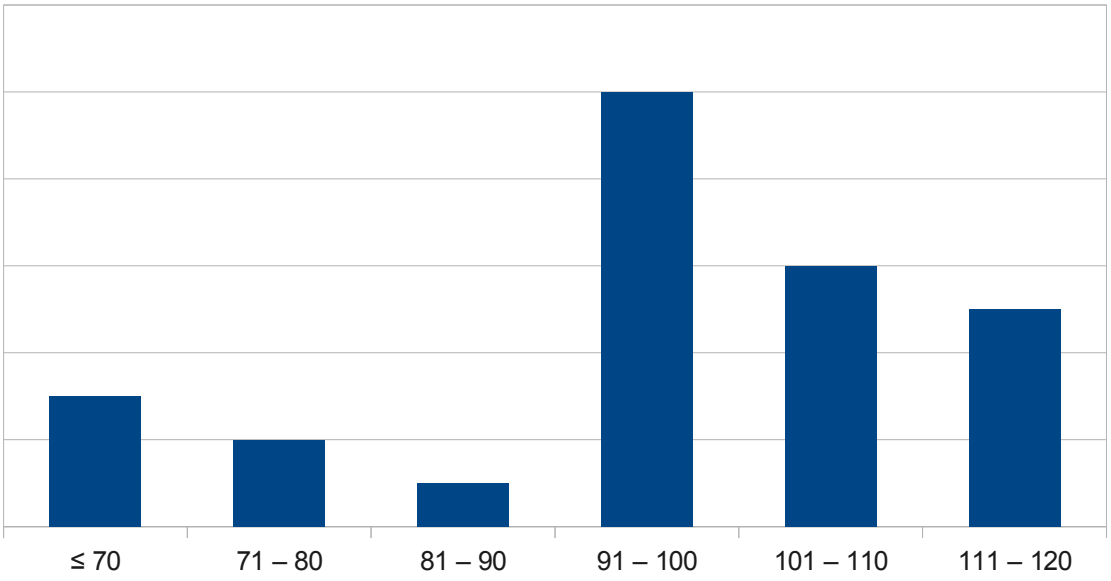
CS143 Midterm Exam Solutions

The distribution of the exam grades was as follows:



Overall, the final statistics were as follows:

Mean: 96 / 120

Median: 100 / 120

Standard Deviation: 17

We are **not** grading this course using raw point totals and will instead be grading on a (fairly generous) curve. Roughly speaking, the median score corresponds to roughly a B/B+. If you scored 85 or lower, we suggest getting in touch with us. As always, if you have any comments or questions about the midterm or your grade on the exam, please don't hesitate to drop by office hours! You can also email the staff list with questions.

If you think that we made any mistakes in our grading, please feel free to submit a regrade request to us. Just write a short (one-paragraph or so) description of what you think we graded incorrectly, staple it to the front of your exam, and hand your exam to Jinchao or to Keith by Monday, August 6 at 4:00 PM. We reserve the right to regrade your entire exam if you submit it for a regrade.

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.

No such input exists. In order for the input to produce 1 at the start, the first two characters must be "aa". Now consider the next character. If it's an 'a,' then, using maximal-munch, the scanner would match rule 2 over rule 1, so 1 wouldn't be printed out. If it's a 'b,' the same is true. If it's anything else, then rule 1 will match "aa," but the next character won't match any of the previous rules, so a 4 will be printed out instead of a 2.

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

One possible input is bbbabaa. Maximal-munch tokenizes this as bb, bab, aa, which prints 321.

Problem Two: LL(1) Parsing (30 points total)

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

Addr → *Name* `@` *Name* `. id`

Name → `id` | `id .` *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.

Here is one grammar:

Addr → ***Name*** `@ id .` ***Name***

Name → `id` ***Name'***

Name' → ε | `. id` ***Name'***

A common mistake on this question was to left-factor the *Name* nonterminal productions, but to forget to adjust the initial rule. If you don't interchange the last symbols of the first production, there will be a FIRST/FOLLOW conflict with the *Name'* rule.

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

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

For our grammar, the FIRST sets are

FIRST(*Addr*) = { `id` }

FIRST(*Name*) = { `id` }

FIRST(*Name'*) = { `.`, ε }

The FOLLOW sets are

FOLLOW(*Addr*) = { `$` }

FOLLOW(*Name*) = { `@`, `$` }

FOLLOW(*Name'*) = { `@`, `$` }

(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 drawing the table, we've already provided the columns in the table. You just need to provide the rows.

For our grammar, the table is as follows

	`id`	**`.`**	**`@`**	**`$`**
Addr	*Name* **`@id.`** *Name*			
Name	**`id`** *Name'*			
Name'		**`.`** **`id`** *Name'*	**ε**	**ε**

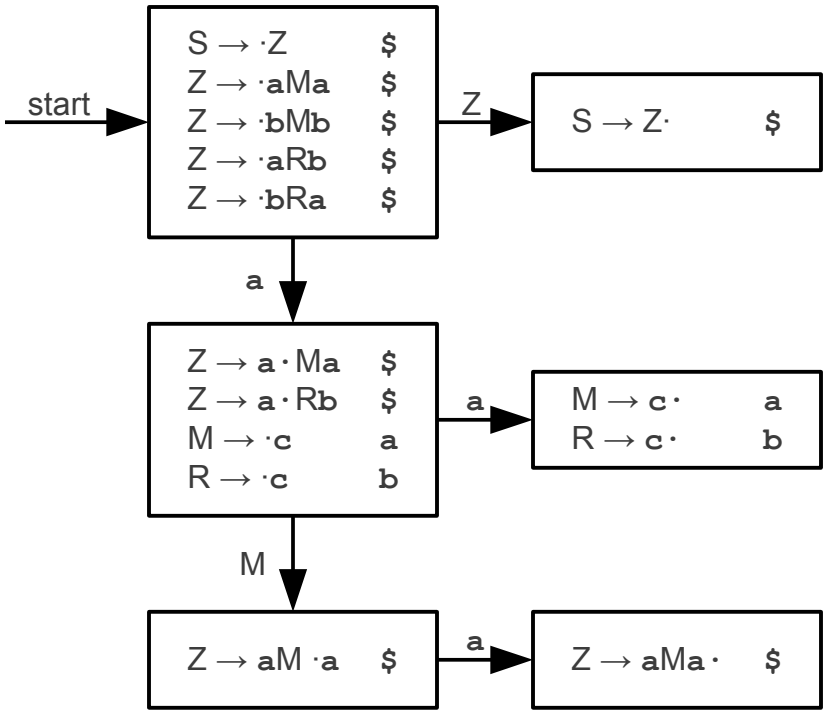
Problem 3: LR Parsing (50 Points Total)

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 (14 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.

We can decide that this grammar is not SLR(1). The LR(1) state

M → `c` · `a`

R → `c` · `b`

Corresponds to the SLR(1) state

M → `c` · FOLLOW(M)

R → `c` · FOLLOW(R)

Since FOLLOW(M) = FOLLOW(R) = { `a`, `b` }, this means that we have a reduce/reduce conflict in this state when using an SLR(1) parser. Thus the grammar is not SLR(1).

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.

We cannot decide whether this grammar is LALR(1). Without seeing either the entire LR(0) automaton or the entire LR(1) automaton, we can't construct the LALR(1) lookaheads.

Some of you noticed that the grammar displays symmetries between `a` and `b`, and used this to conclude (correctly) that the grammar is not LALR(1). We didn't require you to answer this way, though.

(iii) Parsing Efficiency **(20 Points)**

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

$$S \rightarrow A$$
$$A \rightarrow A\mathtt{a} \mid \varepsilon$$

and

$$S \rightarrow A$$
$$A \rightarrow \mathtt{a}A \mid \varepsilon$$

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 $\mathtt{a}^n$, 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 $\mathtt{a}^n$. 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.

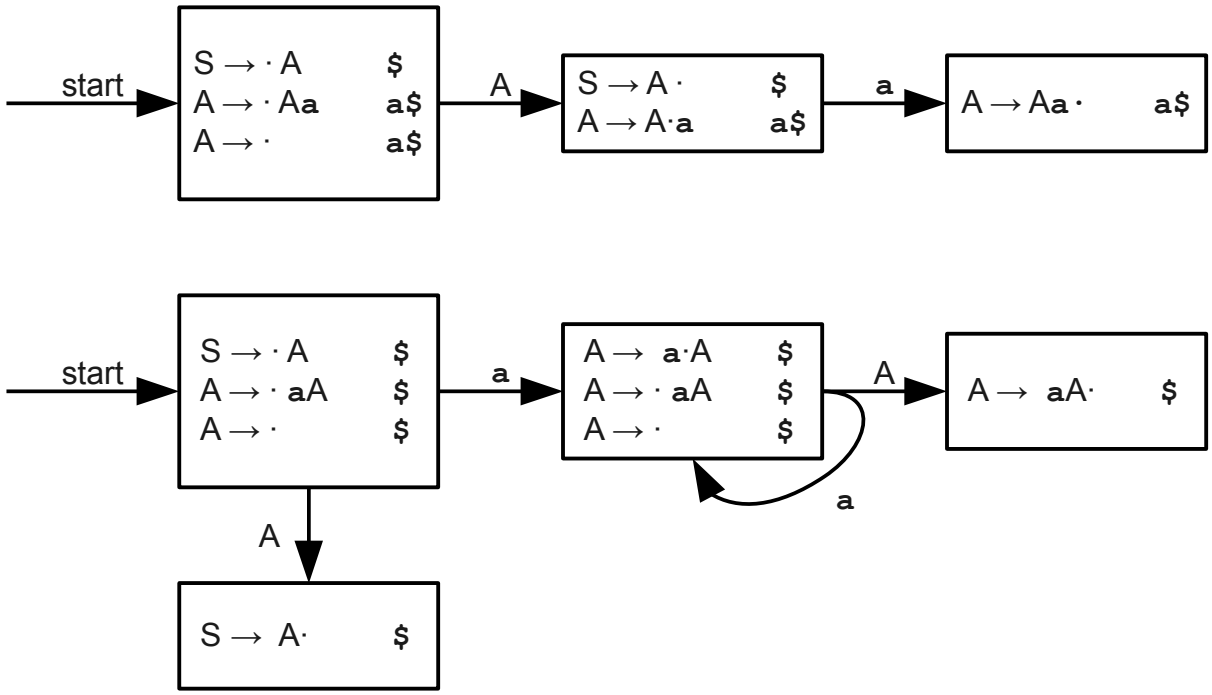
The first grammar uses O(1) stack space, and the second O(*n*). There are many ways that you can justify this; here are a few.

(1) An SLR(1) parser traces out a rightmost derivation in reverse. If we try deriving $\mathtt{a}^n$ using both grammars, we get the following two rightmost derivations:

S	S
A**a**	**a**A
A**aa**	**aa**A
A**aaa**	**aaa**A
A**aaaa**	**aaaa**A
…	…

Since in a shift/reduce parse all reductions occur at the top of the parsing stack, this means that in the case of the first grammar the parsing stack never has more than two symbols on it, because the reduction performed is A → Aa**. If there were more than two symbols on the parsing stack, we couldn't do this reduction. The second grammar, on the other hand, does the reduction A → **a**A, meaning that there will be a string of O(*n*) **a**'s on the parsing stack, followed by the A for which the reduction is done.**

(2) We could construct the parsing automata for these grammars:



Notice that in the the automaton for the first grammar, there are no cycles. This means that we cannot take more than two steps before ultimately applying a reduction. Moreover, of those transitions, only one of them occurs on a terminal. This means that we will shift at most one terminal before doing a reduction, and since that reduction (A → `Aa`) removes a terminal from the stack, the stack height can't get any higher than O(1).

In the second automaton, however, there is a cycle in the automaton. If we see the string `a`^{n}, we will keep cycling from the second state into itself O(n) times before we are allowed to do the reduction A → ε. This means that our parsing stack will have O(n) symbols on it before the first reduction occurs.

(3) We could note that FOLLOW(A) = { `a`, `$` } in the first grammar, while in the second grammar it's { `$` }. Since SLR(1) parsers always reduce on the FOLLOW set, this means that (for the first grammar) we will always reduce A → `Aa` as soon as it gets on the parsing stack, but for the second grammar we can't do any reductions until the lookahead is `$`, which occurs when the last terminal has been shifted.

(4) We could trace out sample parses for each grammar:

Stack	Input	Action
	aa$	Reduce A → ε
A	**aa$**	Shift
A**a**	**a$**	Reduce A → A**a**
A	**a$**	Shift
A**a**	**$**	Reduce A → A**a**
A	**$**	Reduce S → A
S	**$**	Accept

Notice that the stack never has more than two symbols on it. As soon as we reduce A → ε, the stack will never grow past A`a`**, since whenever we reach A**`a`** we will do a reduction back to A. Thus the stack has height O(1).**

On the other hand, the second parser will operate as follows:

Stack	Input	Action
	aaa$	Shift
a	**aa$**	Shift
aa	**a$**	Shift
aaa	**$**	Reduce A → ε
aaaA	**$**	Reduce A → **a**A
aaA	**$**	Reduce A → **a**A
aA	**$**	Reduce A → **a**A
A	**$**	Reduce S → A
S	**$**	Accept

Here, the only way to remove `a`'s from the stack is to reduce with the rule A → `a`A, but this is only possible once an A is pushed onto the stack, which only happens when we use the rule A → ε, and this can only occur when the lookahead is `$`. Consequently, the parser will have to shift all the a's before it can reduce them, which requires O(*n*) stack space.

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):

$S \rightarrow A$ (1)

$A \rightarrow \texttt{a}BE$ (2)

$B \rightarrow \texttt{b}CD$ (3)

$C \rightarrow \texttt{c}$ (4)

$D \rightarrow \texttt{d}$ (5)

$E \rightarrow \texttt{e}FG$ (6)

$F \rightarrow \texttt{f}$ (7)

$G \rightarrow \texttt{g}$ (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.

An LL(1) parser traces a leftmost derivation. Here is a leftmost derivation of `abcdefg`, along with the rules applied at each step:

S		
⇒ A	**(1)**	**S → A**
⇒ aBE	**(2)**	**A → aBE**
⇒ abCDE	**(3)**	**B → bCD**
⇒ abcDE	**(4)**	**C → c**
⇒ abcdE	**(5)**	**D → d**
⇒ abcdeFG	**(6)**	**E → eFG**
⇒ abcdefG	**(7)**	**F → f**
⇒ abcdefg	**(8)**	**G → g**

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.

An LR(1) parser traces a rightmost derivation in reverse. Here is a rightmost derivation of the string `abcdefg`:

S		
⇒ A	**(1)**	**S → A**
⇒ `aBE`	**(2)**	**A → `aBE`**
⇒ `aBeFG`	**(6)**	**E → `eFG`**
⇒ `aBeFg`	**(8)**	**G → `g`**
⇒ `aBefg`	**(7)**	**F → `f`**
⇒ `abCDefg`	**(3)**	**B → `bCD`**
⇒ `abCdefg`	**(5)**	**D → `d`**
⇒ `abcdefg`	**(4)**	**C → `c`**

Thus the reductions are done in order 4, 5, 3, 7, 8, 6, 2, 1:

C → `c`

D → `d`

B → `bCD`

F → `f`

G → `g`

E → `eFG`

A → `aBE`

S → A

(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:

$$E \rightarrow E \texttt{ + } E$$
$$E \rightarrow E \texttt{ * } E$$
$$E \rightarrow \texttt{int}$$
$$E \rightarrow \texttt{(}E\texttt{)}$$

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.

LL(1) parsers are given just the current nonterminal and one token of lookahead to make the decision about which production to use. Even if the parser knew the precedences of addition and multiplication, when deciding whether to produce E → E + E or E → E * E, the parser only sees the next token of lookahead, meaning that it can't see what operator is being applied. Consequently, even knowing the relative precedence won't help the parser, since the parser can't see far enough ahead into the input to determine which operator is applied first.