Midterm I

- Please read all instructions (including these) carefully.
- There are six questions on the exam, each worth between 15 and 25 points. You have 3 hours 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: Sample Solution

SID or SS#: ____________________

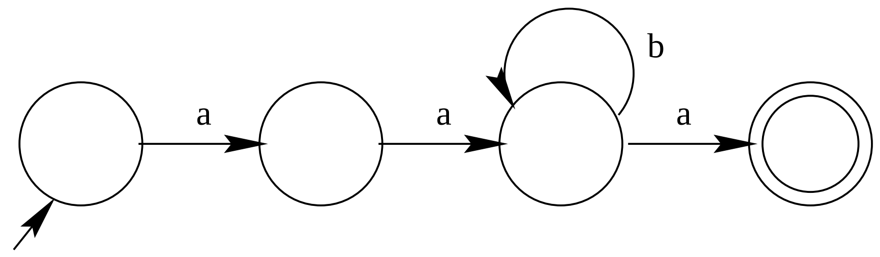
Problem	Max points	Points
1	15	
2	15	
3	25	
4	15	
5	20	
6	20	
TOTAL	110	

1. **Regular Expressions and Finite Automata** (15 points)

   (a) Give a regular expression for the following definition of strings; you may use flex notation in your answer. A string begins and ends with a double-quote. Between the quotes, there may be any sequence of digits and upper- and lower-case letters. It is also permitted to have a “\$” followed immediately by a “\n” (a newline character). No other combination of characters is permitted inside a string.

```
In the following, \n stands for the single newline character:

        ``(A + ... + Z + a + ... + z + 0 + ... + 9 + $\n)*''
```

   (b) Give a DFA accepting the language `aab*a`.



2. **Lexical Analysis** (15 points)

   Consider the following flex-like specification. Parentheses are used to show the association of operations and are not part of the input alphabet.

```
          aa*     { return Token1; }
       c(a|b)*    { return Token2; }
         ab*c     { return Token3; }
          caa*    { return Token4; }
    b*aa*(c|ϵ)    { return Token5; }
```

   (a) Show how the following string is partitioned into tokens. Label each lexeme with the integer of the correct token class. Assume flex semantics for this question and the questions below.

```
abcabcaabbaacccabaccbb
```

```
abc abc aa bbaac c caba c cbb
 3   3   1   5   2  2   2  2
```

   (b) Assuming the input alphabet is $\{a, b, c\}$, are there any non-empty strings that do not match any rule? If so, give an example. If not, explain why.

```
Yes, the string b.
```

   (c) Can any of the rules in this specification be deleted without changing the lexer's behavior on any string? If not, why not? If so, which rules can be removed and why?

```
Yes.  Rule 4 can be removed, because any string it matches is also matched
by Rule 2, which is listed earlier and therefore has higher precedence.
```

3. **Grammars** (25 points)

(a) Give a context-free grammar that generates all regular expressions over the alphabet $\{a, b\}$. Include only standard regular expressions; do not include special flex notation. Write a natural grammar; do not worry about precedence or associativity of operations.

$$\mathtt{S} \rightarrow (\mathtt{S}) \mid \mathtt{S}+\mathtt{S} \mid \mathtt{S}^* \mid \mathtt{SS} \mid \mathtt{a} \mid \mathtt{b} \mid {'\epsilon'}$$

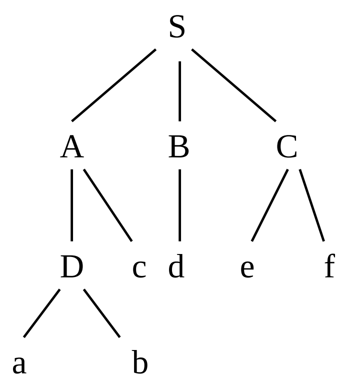
(b) Give an example of a grammar that is unambiguous, left-factored, and not left recursive that is also not LL(1).

```
The terminals are a, b, and c.

S -> A | B
A -> ab
B -> ac

This grammar is not LL(1) because it is not possible to decide with 1 token
of look-ahead which production to expand for S.  It is easy to ses that the
grammar is left factored, unambiguous, and not left recursive.
```

(c) Give the order in which a top-down parser commits to the productions in the parse tree below. Next, give the order in which a bottom-up parser commits to the productions in this tree. Your answer should be a sequence of non-terminals; show the non-terminal expanded/reduced at each step.

Top-down order: `SADBC`

Bottom-up order: `DABCS`



(d) The Cool grammar has a number of ambiguities, all but one of which are introduced by arithmetic operations. The exception is the `let` expression. Give an example illustrating the ambiguity associated with `let`.

```
The expression

   let ... in 1 + 2

can be parsed in two ways:

   (let ... in 1) + 2
or
   let ... in (1 + 2)
```

(e) Consider the following grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{AAAAA} \\ \texttt{A} &\rightarrow \texttt{b}|\texttt{c} \end{aligned}$$

How many derivations does a string in this language have? Justify your answer.

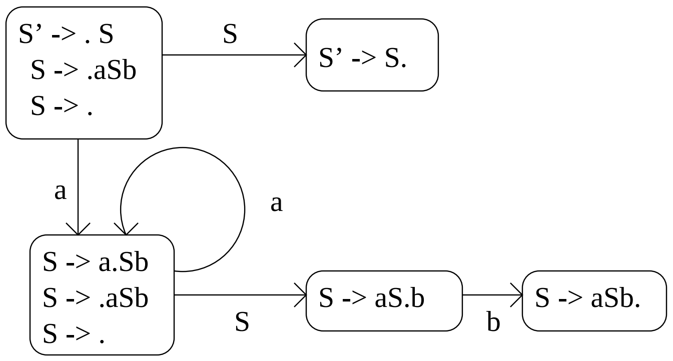
```
The choice of string does not matter; all strings in this language
have the same number of derivations. Consider the string bbbbb.  The
first production is clearly

   S -> AAAAA

For the second production, we can choose A -> b in 5 different ways,
for the third production we will have 4 choices, for the fourth
production we will have 3 choices, and so on.  The total number of
derivations is therefore 5!, or 120.
```

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

Give the DFA of LR(0) items recognizing viable prefixes for the following grammar. Do not add an extra start symbol to the grammar. Is the grammar SLR(1)? Justify your answer.

$$\begin{aligned} \mathtt{S}' &\rightarrow \mathtt{S} \\ \mathtt{S} &\rightarrow \mathtt{aSb} \mid \epsilon \end{aligned}$$

The DFA is given below. There are only two states with multiple items, and the only reduce move in these states is S $\rightarrow$. if the lookahead is b or \$ (i.e., Follow(S) $= \{\mathtt{b}, \$\}$). The only shift moves in these states are on input a. Therefore there are no conflicts and the grammar is SLR(1).



5. **Syntax-Directed Translation** (20 points)

The designers of FOOL (the Failed Object-Oriented Language) thought it would be nice to have two kinds of parentheses (...) and [...] in the language. The idea is that it is easier to see the association of expressions with nested parentheses, such as $((3+1)*5)$, if one can use two different parenthesis notations, such as $[(3+1)*5]$. A grammar for a fragment of FOOL is:

$$\begin{array}{lcl} \texttt{E} & \rightarrow & \texttt{int} \\ & | & [\texttt{E}] \\ & | & (\texttt{E}) \\ & | & \texttt{E} + \texttt{E} \end{array}$$

To make best use of this feature, expressions should strictly alternate between the [...] and (...) parentheses. An expression is *strict* if one of the following is true:

(a) An integer is always strict.

(b) $\texttt{E}_1 + \texttt{E}_2$ is strict if both $\texttt{E}_1$ and $\texttt{E}_2$ are strict.

(c) [E] is strict if two conditions hold. First, E is strict. Second, if E has parenthesized subexpressions, then all outer-most parentheses inside E are (...).

(d) (E) is strict if two conditions hold. First, E is strict. Second, if E has parenthesized subexpressions, then all outer-most parentheses inside E are [...].

If none of (a)-(d) is true for an expression E, then E is not strict.

Write a syntax-directed translation that assigns an attribute E.strict the value *true* if E is strict and *false* otherwise. You may use attributes in addition to strict if you like.

```
We use two extra boolean attributes ``paren'' and ``bracket'' to track whether
an expression has outermost round parens, square brackets, both, or neither.
```

$$\begin{array}{lcll} \texttt{E} & \rightarrow & \texttt{int} & \texttt{E.strict = true} \\ & & & \texttt{E.paren = false} \\ & & & \texttt{E.brack = false} \\ \\ & | & [\texttt{E}_1] & \texttt{E.strict} = \texttt{E}_1\texttt{.strict} \wedge \neg \texttt{E}_1\texttt{.brack} \\ & & & \texttt{E.paren} = \texttt{false} \\ & & & \texttt{E.brack} = \texttt{true} \\ \\ & | & (\texttt{E}_1) & \texttt{E.strict} = \texttt{E}_1\texttt{.strict} \wedge \neg \texttt{E}_1\texttt{.paren} \\ & & & \texttt{E.paren} = \texttt{true} \\ & & & \texttt{E.brack} = \texttt{false} \\ \\ & | & \texttt{E}_1 + \texttt{E}_2 & \texttt{E.strict} = \texttt{E}_1\texttt{.strict} \wedge \texttt{E}_2\texttt{.strict} \\ & & & \texttt{E.paren} = \texttt{E}_1\texttt{.paren} \vee \texttt{E}_2\texttt{.paren} \\ & & & \texttt{E.brack} = \texttt{E}_1\texttt{.brack} \vee \texttt{E}_2\texttt{.brack} \end{array}$$

6. **Parsing Tables** (20 points)

   Below are the "action" and "goto" tables for an LR parser. The "goto" table includes only moves of the parsing automaton on non-terminals; the moves on terminals are encoded in the shift moves of the "action" table. The actions should be interpreted as follows:

   - $s(n)$ shifts the input and goes to state n.
   - $r(n, T)$ pops n elements off of the stack pushes the non-terminal T onto the stack. An r-action is a reduce move, given in a non-standard way.
   - *acc* means accept.
   - A blank is an error entry.

   The non-terminals of the grammar from which these tables were generated are A, B, and C. No two productions for A have the same number of symbols on the right-hand side; similarly, all productions for B and C have different lengths.

   What is the grammar from which these tables were produced?

State	*action*						*goto*		
	a	b	c	d	e	$	A	B	C
0	s(5)			s(4)			1	2	3
1		s(6)				acc			
2		r(1,A)	s(7)		r(1,A)	r(1,A)			
3		r(1,B)	r(1,B)		r(1,B)	r(1,B)			
4	s(5)			s(4)			8	2	3
5		r(1,C)	r(1,C)		r(1,C)	r(1,C)			
6	s(5)			s(4)				9	3
7	s(5)			s(4)					10
8		s(6)			s(11)				
9		r(3,A)	s(7)		r(3,A)	r(3,A)			
10		r(3,B)	r(3,B)		r(3,B)	r(3,B)			
11		r(3,C)	r(3,C)		r(3,C)	r(3,C)			

```
The grammar is

A -> A b B
A -> B
B -> B c C
B -> C
C -> d A e
C -> a

The essence of the problem is to discover what can be on the stack when a
reduction is about to happen.  One way to solve the problem is to reconstruct
the parsing DFA from the table and read the moves.  A simpler way is to reason
as follows.  In state 9 there is a reduce move r(3,A), so we know there is a
```

production $A \rightarrow XYZ$ for some X, Y, and Z. How could the DFA get into state 9? It could get there from a ''goto B'' out of state 6. Therefore, Z = B. One way to get to state 6 is via a ''shift b'' action from state 8, so Y = b. State 8 is reached from a goto on a reduce to A, so X = A and the production is $A \rightarrow AbB$.

There are alternatives in this line of reasoning. For example, one can get to state 6 either via a shift in state 8 or a shift in state 1. However, the problem statement says that there is only one production for A with a rhs of length 3, so which alternative we select cannot make a difference in the answer (it is also easy to check that choosing state 1 instead of state 8 leads to the same answer).

The reasoning for the other productions is similar.