CS143 Midterm
Spring 2026

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 80 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason other than to access the class webpage.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- 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.

SUNET ID: ____

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	10	
2	20	
3	20	
4	25	
5	25	
TOTAL	100	

1. Lexical Analysis

Consider the following flex-like lexical specification:

```
(ab)+     {print "1"}
a*ba*     {print "2"}
b+        {print "3"}
(ab)*a    {print "4"}
```

Given the following input string:

ababaabbbaba

What does the lexer print?

Answer:

4 1 3 2

2. Context-Free Grammars

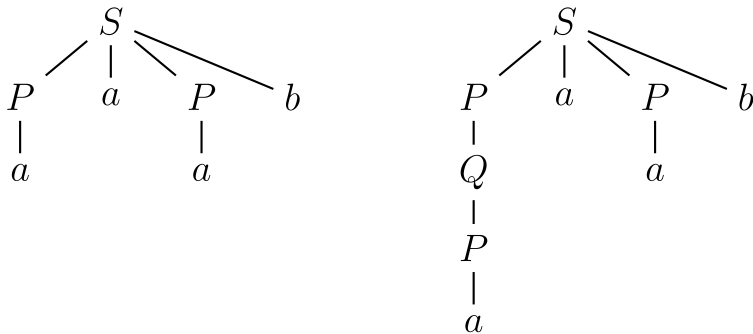
Consider the following grammar G on the alphabet $\Sigma = \{a, b\}$:

$$S \rightarrow PaPb \mid QbQa$$
$$P \rightarrow Q \mid a$$
$$Q \rightarrow P \mid b$$

(a) Is the grammar ambiguous? Justify your answer.

Answer:

Yes, the grammar is ambiguous. Here are two different parse trees for the same input:



(b) Describe the language $L(G)$ that the CFG generates.

Answer:

The language $L(G)$ consists of the following eight strings:

```
aaab    aabb    baab    babb
abaa    abba    bbaa    bbba
```

(c) Is the language $L(G)$ described by the CFG a regular language? Justify your answer.

Answer:

Yes, a finite number of strings is regular. For example, we can take their union.

3. Semantic Actions

Consider boolean expressions involving $\wedge$ (and), $\vee$ (or), $\neg$ (not) and variables. We say an expression is normalized if the $\neg$ operator is applied only to variables, not expressions (be it an $\wedge$, $\vee$, or $\neg$ expression). For example, $(x \wedge \neg y) \vee z$ is normalized, while $\neg(x \wedge y)$ is not.

We can normalize a boolean expression by repeatedly pushing down the $\neg$ operator:

$$
\begin{aligned}
\neg(E_1 \wedge E_2) &= \neg E_1 \vee \neg E_2 \\
\neg(E_1 \vee E_2) &= \neg E_1 \wedge \neg E_2 \\
\neg\neg x &= x
\end{aligned}
$$

Your goal is to parse an input boolean expression that is not normalized, normalize it using semantic actions, and produce an AST that represents the normalized expression.

For your convenience, we have provided the relevant AST definitions and APIs for you:

```
enum NodeType { And, Or, Not, Var };

struct Node {

  NodeType getType();

  Node* getLhs(); // requires getType() == And || getType() == Or

  Node* getRhs(); // requires getType() == And || getType() == Or

  Node* getOperand(); // requires getType() == Not

  std::string getName(); // requires getType() == Var

  ...

};

// APIs to create the AST nodes

Node* make_variable(std::string name);

Node* make_and(Node* lhs, Node* rhs);

Node* make_or(Node* lhs, Node* rhs);

Node* make_not(Node* var); // requires var->getType() == Var
```

Note that since we require the AST to represent normalized expressions, the $\neg$ operator can only be applied on variables. Thus, `make_not` API requires that `var->getType() == Var`.

The CFG and associated semantic actions are defined as below:

```
E → E ∨ F { $$ = make_or($1, $2); }
   | F     { $$ = $1; }
F → F ∧ G { $$ = make_and($1, $2); }
   | G     { $$ = $1; }
G → ¬G     { $$ = handle_not($1); }
   | H     { $$ = $1; }
H → (E)    { $$ = $1; }
   | id    { $$ = make_variable($1); }
```

Implement the function `Node* handle_not(Node* e)` so that the semantic actions normalize the input expression. For the purpose of this problem, you can ignore memory leaks.

Answer:

```
Node* handle_not(Node* e) {
  switch (e->getType()) {
    case And:  return make_or(handle_not(e->getLhs()), handle_not(e->getRhs()));
     case Or:  return make_and(handle_not(e->getLhs()), handle_not(e->getRhs()));
    case Not:  return e->getOperand();
    case Var:  return make_not(e);
  };
}
```

4. Top-Down Parsing

We have the following *partially-filled* LL(1) parse table for an unknown grammar G. The terminals are $\{a, b, c, d\}$ and the non-terminals are $\{S, A\}$ where S is the start symbol. A dash (—) means the cell is blank (error) and a question mark (?) means the cell's contents have been erased and we want to recover them.

	a	b	c	d	\$
S	$A\,b$	$?_b$	$?_c$	d	—
A	$a\,A$	ε	c	—	—

We can assume that the table was constructed correctly from G using the algorithm from lecture, and that G is LL(1).

(a) Using the table, determine Follow(S) and Follow(A). Briefly justify each.

Answer:

Follow(S) $= \{\,\$\,\}$, since S is the start symbol.
Follow(A) $= \{\,b\,\}$. Since $T[A, b] = \varepsilon$, we can say that $b \in$ Follow(A).

(b) Provide all productions of A and give a brief justification using the table.

Answer:

Each distinct entry in row A corresponds to one production of A. The entries are $a\,A$, ε, and c, so

$$A \;\rightarrow\; a\,A \;\mid\; c \;\mid\; \varepsilon$$

(c) What production must fill cell $?_c$ (the $[S, c]$ entry)? Give an explanation *without* building the full parse-table.

Answer:

$$?_c = A\,b$$

We know $S \to A\,b$ is a production (from $T[S, a]$), and the production $A \to c$ makes $c \in \text{First}(A\,b)$.

(d) What production must fill cell $?_b$ (the $[S, b]$ entry)? Write down a brief explanation also.

Answer:

$$?_b \;=\; A\,b$$

Since $A \to \varepsilon$, so the right-hand side $A\,b$ can derive b directly. Hence $b \in \text{First}(A\,b)$.

5. Bottom-Up Parsing

At the bottom of the page is the skeleton of a DFA parsing automaton for recognizing viable prefixes, showing just the states and transitions.

(a) Write a grammar that results in this parsing automaton. Hint: the grammar needs only two productions.

Answer:

$S \rightarrow (S) \mid ab$

(b) Fill in the states with the LR(0) items from your grammar and label each edge with the correct transition symbol. There should be no shift-reduce or reduce-reduce conflicts in your automaton (under LR(0) parsing rules).

Answer: