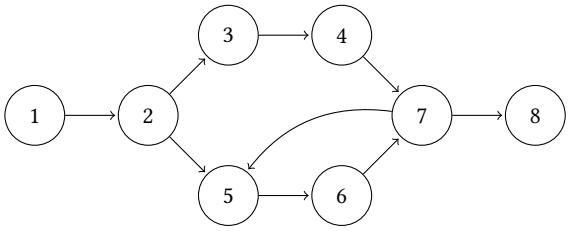
Mid-term Exam (COSE 312 Compilers, Spring 2025)

Student Number: **Name:**

Problem 1. (5pts) Suppose a directed graph, $G = (N, E)$, is given, where N is the set of nodes and $E \subseteq N \times N$ the set of edges. Given G, let $\mathsf{Reach}(I)$ be the set of nodes reachable from an initial set I of nodes. For example, for the following graph with $N = \{1,2,3,4,5,6,7,8\}$ and $E = \{(1,2), (2,3), (2,5), (3,4), (4,7), (5,6), (6,7), (7,5), (7,8)\}$, $\mathsf{Reach}(\{3,6\}) = \{3,4,5,6,7,8\}$.



$\mathsf{Reach}(I)$ can be defined as the least fixed point of a function $F : 2^N \to 2^N$. Complete the definition of F:

$$F(X) = I \cup \boxed{}$$

$$F(X) = I \cup \{m \mid n \in X, (n,m) \in E\}$$

Problem 2. (10pts) Consider the following grammar for lists:

$$E \to [L] \mid a, \quad L \to E\,T, \quad T \to ;L \mid \epsilon$$

E denotes a single element, either an atom a or a bracketed list $[L]$. L is a list, which consists of an element and a tail. T is a tail, either empty or a semicolon followed by a list. E is the start variable.

(1) Compute the *FIRST* and *FOLLOW* sets:
- *FIRST*(E): $\{[, a\}$
- *FIRST*(L): $\{[, a\}$
- *FIRST*(T): $\{;, \epsilon\}$
- *FOLLOW*(E): $\{\$, ;,]\}$
- *FOLLOW*(L): $\{]\}$
- *FOLLOW*(T): $\{]\}$

(2) Construct the LL parsing table for the grammar:

	a	[	]	;	$
E					
L					
T					

	a	[	]	;	$
E	$E \to a$	$E \to [L]$			
L	$L \to ET$	$L \to ET$			
T			$T \to \epsilon$	$T \to ;L$	

Problem 3. (15pts) Consider the grammar written in yacc:

```
program : cmd EOF  { $1 }
cmd : IDENT ASSIGN aexp SEMICOLON { Assn ($1, $3) }
| SKIP SEMICOLON { Skip }
| cmd cmd { Seq ($1, $2) }
| IF LPAREN bexp RPAREN LBRACE cmd RBRACE
  ELSE LBRACE cmd RBRACE { If ($3, $6, $10) }
```

Is this grammar ambiguous? If so, (1) identify which production rules cause which types of conflicts, (2) give and explain an example program that have multiple parse trees, and (3) rewrite the grammar to eliminate these conflicts.

(1) The rule "`cmd: cmd cmd`" causes shift/reduce conflicts:
`cmd . cmd (shift)`
`cmd cmd . (reduce)`

(2) Command "`skip; skip; skip;`" produces two parse trees: `skip; (skip; skip;)` and `(skip; skip;) skip;`.

(3) Rewrite the grammar as follows:

```
program : cmd_seq EOF { $1 }
cmd_seq :
| cmd { $1 }
| cmd_seq cmd { Seq ($1, $2) }
cmd : IDENT ASSIGN aexp SEMICOLON {Assn ($1, $3)}
| SKIP SEMICOLON { Skip }
| IF LPAREN bexp RPAREN LBRACE cmd_seq RBRACE
ELSE LBRACE cmd_seq RBRACE { If ($3, $6, $10) }
```

Problem 4. (10pts) Consider the "dangling-else" grammar:

$$\begin{array}{lllll} (1) & S & \to & i\,S\,e\,S \\ (2) & S & \to & i\,S \\ (3) & S & \to & a \end{array}$$

The SLR parsing is as follows:

STATE	i	e	a	$	S
0	$s2$		$s3$		$g1$
1				acc	
2	$s2$		$s3$		$g4$
3		$r3$		$r3$	
4		**s5, r2**		$r2$	
5	$s2$		$s3$		$g6$
6		$r1$		$r1$	

(1) In the $(4, e)$ entry, there is a shift/reduce conflict. Which action is correct? Explain why and remove the ambiguity in the parsing table.
Shift is correct because the `else` should be matched with the most recent `if`. Therefore, remove $r2$ from the entry.

(2) With the resulting parsing table, complete the parsing actions on input *iiaea*:

Step	Stack	Symbols	Input	Action
1	0		*iiaea*\$	shift
2				
3				
4				
5				
6				
7				
8				
9				
10				

Stack	Symbols	Input	Action
0		*iiaea*\$	shift
0 2	*i*	*iaea*\$	shift
0 2 2	*ii*	*aea*\$	shift
0 2 2 3	*iia*	*ea*\$	reduce by $S \to a$
0 2 2 4	*iiS*	*ea*\$	shift
0 2 2 4 5	*iiSe*	*a*\$	shift
0 2 2 4 5 3	*iiSea*	\$	reduce by $S \to a$
0 2 2 4 5 6	*iiSeS*	\$	reduce by $S \to iSeS$
0 2 4	*iS*	\$	reduce by $S \to iS$
0 1	*S*	\$	accept

Problem 5. (10pts) Consider the statements of **While**:

$$S \quad \to \quad x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ S_1\ S_2 \mid \texttt{while}\ b\ S$$

In class, we have defined its big-step ($\langle S, s\rangle \to s'$) and small-step ($\langle S, s\rangle \Rightarrow \gamma$, where γ is either $\langle S', s'\rangle$ or s') operational semantics. We would like to extend the language with the do-while statement: do S `while` b.

(1) Define the big-step semantics rule for do S `while` b.
Two possible solutions:
A. Single rule:

$$\frac{\langle S, s\rangle \to s' \quad \langle \texttt{while}\ b\ S, s'\rangle \to s''}{\langle \texttt{do}\ S\ \texttt{while}\ b, s\rangle \to s''}$$

B. Two rules:

$$\frac{\langle S, s\rangle \to s'}{\langle \texttt{do}\ S\ \texttt{while}\ b, s\rangle \to s'} \ \mathcal{B}[\![\ b\ ]\!](s') = \mathit{false}$$

$$\frac{\langle S, s\rangle \to s' \quad \langle \texttt{do}\ S\ \texttt{while}\ b, s'\rangle \to s''}{\langle \texttt{do}\ S\ \texttt{while}\ b, s\rangle \to s''} \ \mathcal{B}[\![\ b\ ]\!](s') = \mathit{true}$$

(2) Define the small-step semantics rule for do S `while` b.

$$\frac{}{\langle \texttt{do}\ S\ \texttt{while}\ b, s\rangle \Rightarrow \langle S; \texttt{while}\ b\ S, s\rangle}$$

Problem 6. (10pts) Consider the following property between the big-step ($\to$) and small-step ($\Rightarrow$) operational semantics:

$$\forall S \in \textbf{Stm}, s, s' \in \textbf{State}.\ \langle S, s\rangle \to s' \implies \langle S, s\rangle \Rightarrow^* s'.$$

(1) Explain what this property means.
If the big-step evaluation of S starting in state s terminates in state s', then the small-step semantics starting from the same configuration also terminates in state s'. Conversely, if the small-step evaluation does not terminate, then the big-step evaluation also does not terminate.

(2) In class, we proved this property by induction on the derivation of $\langle S, s\rangle \to s'$. Explain why we cannot prove it by structural induction on S.
Because the big-step semantics for the while statement

$$\frac{\langle S, s\rangle \to s'' \quad \langle \texttt{while}\ b\ S, s''\rangle \to s'}{\langle \texttt{while}\ b\ S, s\rangle \to s'} \ \text{if } \mathcal{B}[\![\ b\ ]\!](s) = \mathit{true}$$

involves a recursive reference to the same while statement in the premise.

Problem 7. (10pts) Consider the simple expressions:

$$E \to n \mid -E \mid E + E$$

Let us translate the expressions into a stack machine M. The machine code (C) is defined as follows:

$$c \in C \to \epsilon \mid \texttt{push}(n) :: C\ (n \in \mathbb{Z}) \mid \texttt{add} :: C \mid \texttt{rev} :: C$$

A configuration consists of an instruction and a stack:

$$\langle c, s\rangle \in C \times \mathbb{Z}^*$$

E.g., $\texttt{push}(1) :: \texttt{push}(2) :: \texttt{add} :: \texttt{rev}$ is run as follows:

Stack	Code
ϵ	$\texttt{push}(1) :: \texttt{push}(2) :: \texttt{add} :: \texttt{rev}$
1	$\texttt{push}(2) :: \texttt{add} :: \texttt{rev}$
$2 :: 1$	$\texttt{add} :: \texttt{rev}$
3	$\texttt{rev}$
-3	

(1) The semantics of M is defined by small-step operational semantics. Define the transition relation:

$$(\triangleright) \subseteq (C \times \mathbb{Z}^*) \times (C \times \mathbb{Z}^*).$$

$$\begin{array}{rcll} \langle \texttt{push}(n).C, s\rangle & \triangleright & \langle C, n :: S\rangle & \\ \langle \texttt{add} :: C, n_1 :: n_2 :: S\rangle & \triangleright & \langle C, n :: S\rangle & (n = n_1 + n_2) \\ \langle \texttt{rev} :: C, n :: S\rangle & \triangleright & \langle C, -n :: S\rangle & \end{array}$$

(2) Define the function $\mathcal{T}$ that translates E to C:

$$\mathcal{T} : E \to C$$

$$\begin{array}{rcl} \mathcal{T}(n) & = & \texttt{push}(n) \\ \mathcal{T}(-E) & = & \mathcal{T}(E) :: \texttt{rev} \\ \mathcal{T}(E_1 + E_2) & = & \mathcal{T}(E_1) :: \mathcal{T}(E_2) :: \texttt{add} \end{array}$$

Problem 8. (30pts) True/False questions: Each correct answer earns 5 points, and each incorrect answer causes you to lose 5 points. Do not answer if you are uncertain.

(1) Every parse tree has a unique leftmost derivation. O
(2) For any $k \geq 0$, LR(k) parsing is strictly more powerful than LL(k) parsing. O

(3) In the **While** language below, every syntactically correct program is also semantically correct (i.e., no undefined semantics). O

$a \rightarrow n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$

$b \rightarrow \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$

$S \rightarrow x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ S_1\ S_2 \mid \texttt{while}\ b\ S$

(4) In the simplified S language below, every syntactically correct program is also semantically correct (i.e., no undefined semantics). X

$S \rightarrow lv := e \mid \texttt{if}\ e\ S_1\ S_2 \mid \texttt{while}\ e\ S \mid S_1; S_2$

$lv \rightarrow x \mid x\texttt{[}e\texttt{]}$

$e \rightarrow n \mid lv \mid e\texttt{+}e \mid \texttt{-}e \mid e == e \mid e < e \mid \texttt{!}e \mid e\texttt{||}e \mid e\texttt{\&\&}e$

(5) In LR parsing, the language of handles (i.e., substrings that match the body of a production and whose reduction leads to a right-sentential form) is not regular but context-free. X

(6) In **While**, the semantic function for statements is defined as a total function. X