6.035	**Name:** ____________________
Spring 2022	**MIT Kerb:** ____________________
Quiz 1 Solution	**Time Limit: 50 min**

- **DO NOT open the exam booklet until you are told to begin. You should write your name and kerb id at the top and read the instructions.**

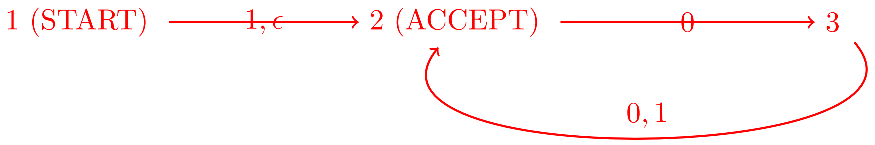
Problem	Points	Score
1	10	
2	8	
3	10	
4	8	
5	9	
6	15	
Total:	60	

1. **Regular Language and DFAs**

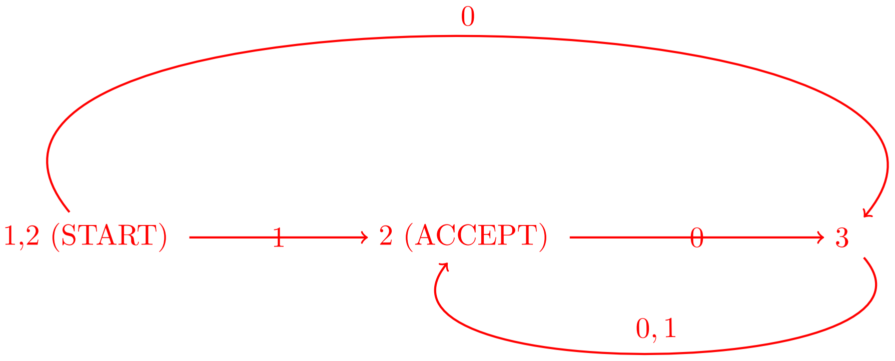
Consider the following regular expression:

$$R = (\epsilon \mid 1)\big(0(0 \mid 1)\big)^*$$

(a) (5 points) Provide an NFA that recognizes the language defined by R. Your NFA should have one start state and one accept state. Identify the start state and the accept state. We advise you to number the states of the NFA.



(b) (5 points) Provide a DFA that recognizes the language defined by R. We advise you to use the product construction presented in class. Your DFA should have one start state and may have multiple accept states. Identify the start state and all accept states.



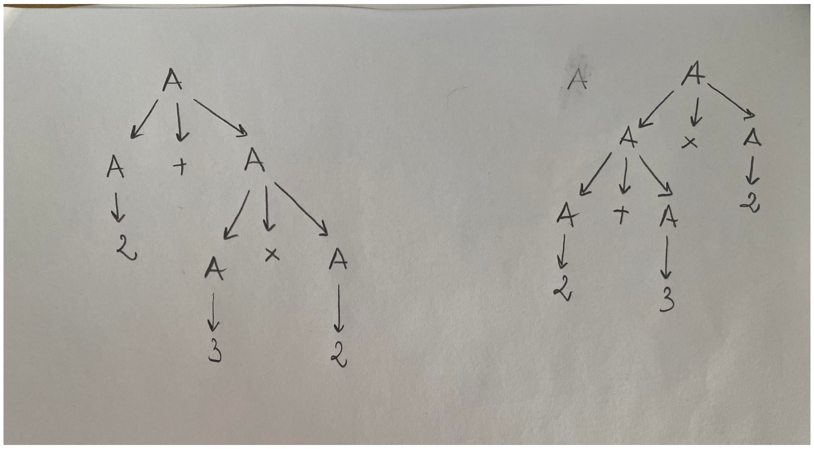
2. **Parse Trees**

This question uses the following grammar

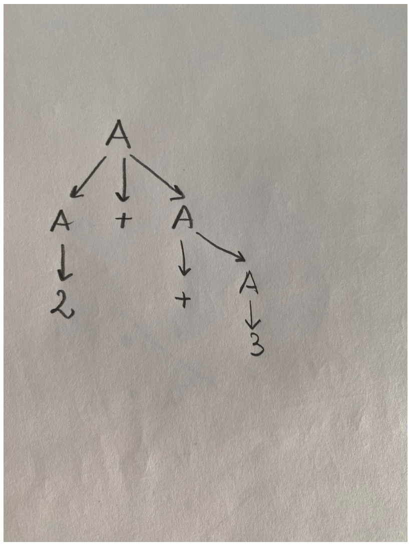
$$\begin{array}{rcl} \textsf{Start} & \rightarrow & A \\ A & \rightarrow & A \times A \\ A & \rightarrow & A + A \\ A & \rightarrow & +A \\ A & \rightarrow & (0|1|2|3) \end{array}$$

For each of the following two expressions, draw all of the parse trees for the expression using the grammar above or state why the expression is not in the language defined by the grammar above.

(a) (4 points) $2 + 3 \times 2$



(b) (4 points) $2 + +3$



3. **Recursive Descent Parser**

Consider the following grammar:

$$\begin{array}{lcl} I & \rightarrow & I\ i\ t \\ I & \rightarrow & I\ i\ t\ e \\ I & \rightarrow & \epsilon \end{array}$$

(a) (5 points) Hack the given grammar to remove left recursion.

$$I \rightarrow i\ t\ I \mid i\ t\ e\ I \mid \epsilon$$

(b) (5 points) We want to parse strings accepted by the above grammar using a Recursive Descent Parser with Predictive Parsing, which uses only a single token lookahead to decide which production rule to apply. Left Factor the grammar from part (a), so that a predictive recursive descent parser can parse it, using a single token lookahead.

$$I \rightarrow i\ t\ E\ I \mid \epsilon$$

$$E \rightarrow e \mid \epsilon$$

4. **Shift Reduce Parsers**

This question uses the following grammar (rules numbered for reference):

$$\begin{array}{rcl} (1)\ \mathsf{Start} & \rightarrow & X\$ \\ (2)\ E & \rightarrow & \mathsf{int} \\ (3)\ E & \rightarrow & (X) \\ (4)\ X & \rightarrow & X\ +\ E \\ (5)\ X & \rightarrow & E \end{array}$$

(a) (2 points) Provide the items for the $X \rightarrow X\ +\ E$ production rule.

$$\begin{array}{rcl} X & \rightarrow & \circ X\ +\ E \\ X & \rightarrow & X \circ\ +\ E \\ X & \rightarrow & X\ +\ \circ\ E \\ X & \rightarrow & X\ +\ E\circ \end{array}$$

(b) (6 points) Provide the Closure of the following sets of items:

- $\mathsf{Closure}(\{X \rightarrow X \circ + E\}) =$
  - $X \rightarrow X \circ\ +\ E$

- $\mathsf{Closure}(\{X \rightarrow X\ +\ \circ E\}) =$
  - $X \rightarrow X\ +\ \circ\ E$
  - $E \rightarrow \circ int$
  - $E \rightarrow \circ(X)$

- $\mathsf{Closure}(\{X \rightarrow X\ +\ E\circ\}) = X \rightarrow X\ +\ E\circ$

5. **OOP & Semantic Analysis**

   The code implements two different graph storage modes (sparse vs. dense graphs). Some implementation specifics are omitted since they are not relevant for completing this problem.

```
class Graph {
    string[] node_properties;

    int get_edge(int row, int col) {
        return 0;
    }

    void set_edge(int row, int col, int value) {}
};

class DenseGraph : Graph {
    int[][] adjacency_matrix;

    int get_edge(int row, int col) {
        return adjacency_matrix[row][col];
    }

    void set_edge(int row, int col, int value) {
        adjacency_matrix[row][col] = value;
    }
};

class SparseGraph : Graph {
    int[] non_zero_values;
    int[] non_zero_columns;
    int[] non_zero_rows;

    int get_edge(int row, int col) {
        ...
    }

    void set_edge(int row, int col, int value) {
        ...
    }

    int get_sparsity() {
        ...
    }
};


class GraphFactory {
    Graph make_graph(bool sparse) {
        if (sparse) {
            return new SparseGraph();
        } else {
            return new DenseGraph();
        }
    }
};
```

(a) (7 points) The following statements use the `Graph` classes defined above. Which of these statements satisfy the relevant semantic checks? Fill in the table below by specifying for each statement whether it satisfies the semantic checks. If it does not, give a brief explanation as to why.

```
1.  DenseGraph g1 = GraphFactory().make_graph(false);

2.  print(GraphFactory().make_graph(false).node_properties);

3.  print(GraphFactory().make_graph(true).get_sparsity());

4.  Graph g2 = SparseGraph();

5.  int e = g2.set_edge(0, 1);

6.  print(SparseGraph().get_sparsity());

7.  print(g2.get_sparsity());
```

Stmt	Semantically Correct? (Yes/No)	Explanation (if not semantically correct)
1	No	return type of method call is not compatible with type of lhs var
2	Yes	
3	No	return type of make graph is Graph, which does not have a get_sparsity
4	Yes	
5	Yes	
6	Yes	
7	No	Compile time type of g2 is Graph, which has no get_sparsity

(b) (2 points) Consider the following code:

```
1.    Graph g1 = Graph();
2.    g1.set_edge(0, 1, 5);
3.    print(g1.get_edge(0, 1));
4.
5.    Graph g = GraphFactory().make_graph(false);
6.    g.set_edge(0, 1, 5);
7.    print(g.get_edge(0, 1));
```

What value is printed when line 3 executes?

What value is printed when line 7 executes?

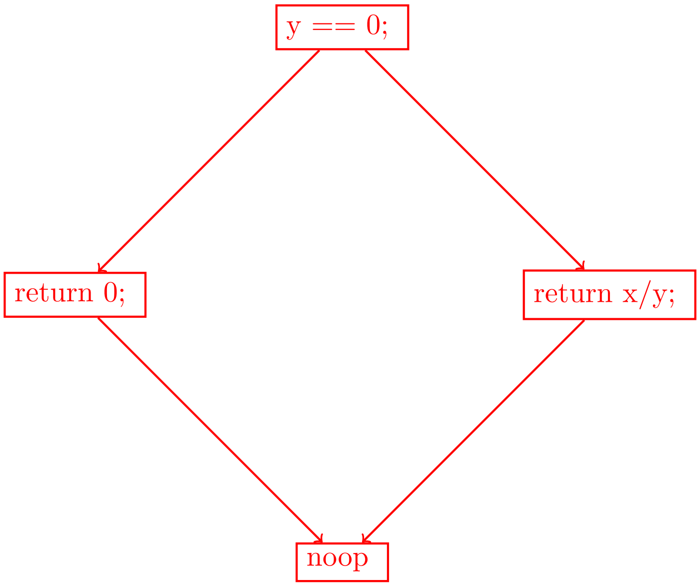
0 on line 3 and 5 on line 7

6. **CFG & Code Generation**

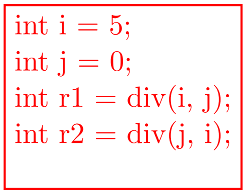
Consider the following 2 procedures (written in C-like language):

```
int div(int x, int y) {
  if (y == 0) {
    return 0;
  } else {
    return x / y;
  }
}

void main() {
    int i = 5;
    int j = 0;
    int r1 = div(i, j);
    int r2 = div(j, i);
}
```

(a) (3 points) Draw the control flow graph for procedure `div`.



(b) (3 points) Draw the control flow graph for procedure `main`.

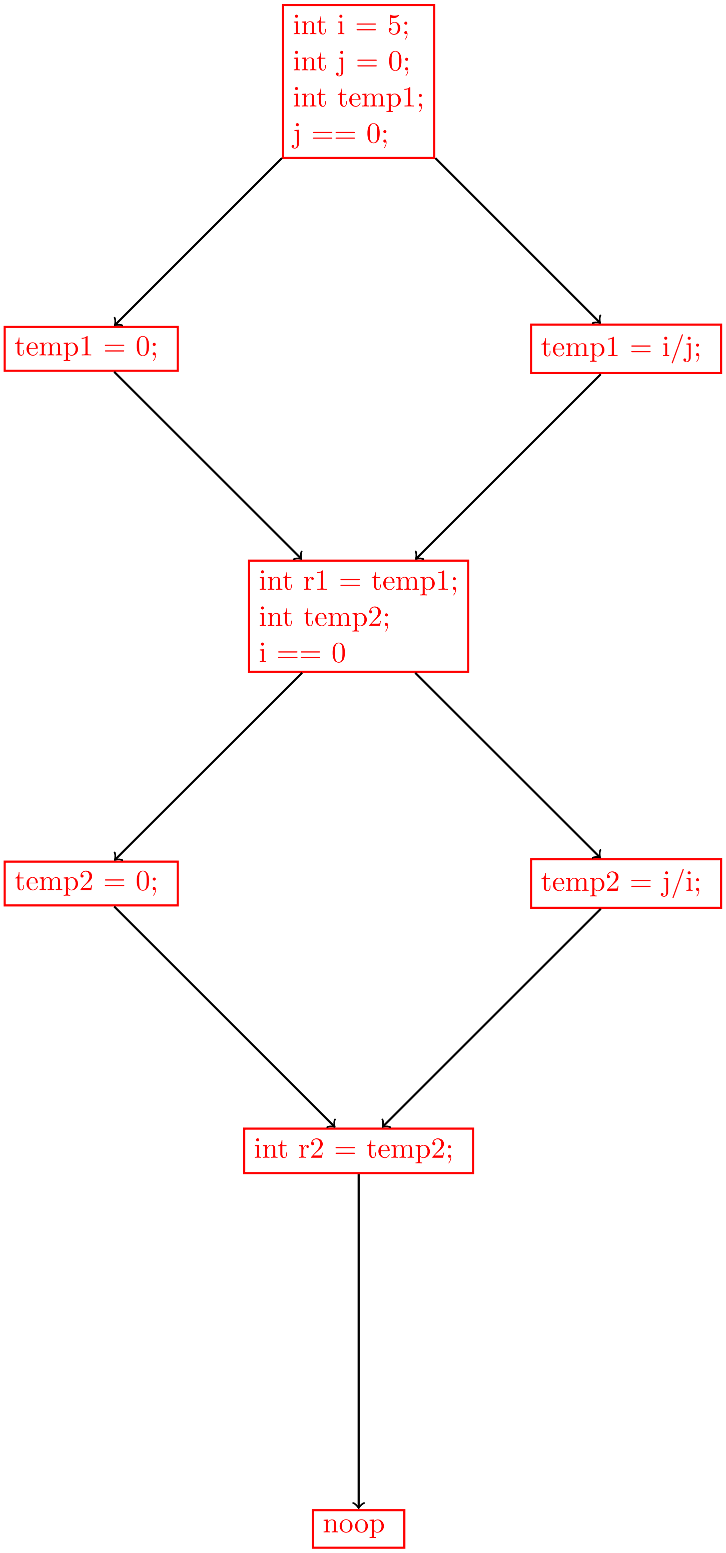


(c) (6 points) You have a smart compiler. It implements an optimization called *inlining*. Inlining eliminates procedure calls by replacing the procedure call with code from the invoked procedure, appropriately replacing the *formal parameters* (the parameters in the procedure definition) with the *actual parameters* (the parameters from the procedure call). This optimization eliminates the procedure call overhead (and may enable additional optimizations).

Rewrite the main function after inlining. We recommend that you introduce temporary variables (t1, t2, ...) as appropriate to hold relevant values after inlining. Make sure that your inlined version preserves the semantics of the original version. In particular, the inlined version should compute the same values of r1 and r2 as the original version.

```
void main() {

    int i = 5;
    int j = 0;

    int temp1 = 0;
    if (j === 0) {
        temp1 = 0;
    } else {
        temp1 = i/j;
    }

    int r1 = temp1                                                           ;

    int temp2=0;
    if (i == 0) {
        temp2 = 0;
    } else {
        temp2 = j/i;
    }

    int r2 =  temp2                                                           ;
}
```

(d) (3 points) Draw the control flow graph for the modified main procedure from part (c).


int i = 5;
int j = 0;
int temp1;
j == 0;
temp1 = 0;
temp1 = i/j;
int r1 = temp1;
int temp2;
i == 0
temp2 = 0;
temp2 = j/i;
int r2 = temp2;
noop