• Print your name and netid, neatly in the space provided above; print your name at the upper right corner of every page. Please print legibly.

• This is a closed book exam. No notes, books, dictionaries, or calculators are permitted.

• Write your answers in the space provided for the corresponding problem. Let us know if you need more paper.

• Suggestions: Read through the entire exam first before starting work. Do not spend too much time on any single problem. If you get stuck, move on to something else and come back later.

• If you run short on time, remember that partial credit will be given.

• If any question is unclear, ask one of us for clarification.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. Recursion (10 points)
Write a tail recursive version of the following function:

```
(define map
  (lambda (f lst)
    (if (eq? lst ()) ()
      (cons (f (car lst)) (map f (cdr lst))))))
```

```
(define map (lambda (f lst) (map-iter f lst ())))
```

```
(define map-iter
  (lambda (f lst r)
    (if (eq? lst ()) (reverse r)
      (map-iter f (cdr lst) (cons (f (car lst)) r))))))
```
2. Data Abstraction (15 points)

(a) Use define-datatype to define a Node datatype for a node in a directed graph. The Node consists of a name, and a list of downstream neighbors, which are nodes reachable via a single edge transition. A list of Nodes is a complete specification for a graph.

Fill in the missing code so that the following works (5 points):
(Note: It is suggested that you read part (b) before filling in the definition.)

(define-datatype Digraph Digraph?
  (Digraph
    (nodes (list-of Node?))))

(define-datatype Node Node?
  (Node
    (name symbol?)
    (nodes (list-of Node?))))

(b) Using the Digraph and Node definitions from above, write a function dfs that will walk the graph, and return a list of names of all reachable nodes in the order they are visited by a depth-first search traversal. (10 points)

(define dfs
  (lambda (digraph start)
    (cases Digraph digraph
      (Digraph (lst) (reverse (dfs-help lst start ()))))))

(define dfs-help
  (lambda (lst start visited)
    (if (member start visited) visited
      (cases Node start
        (Node (s nodes) (dfs-help2 lst nodes (cons start visited)))))))

(define dfs-help2
  (lambda (lst nodes visited)
    (if (eq? nodes ()) visited
      (dfs-help2 lst (cdr nodes) (dfs-help lst (car nodes) visited))))
3. Scoping (15 points)

Write the final output of the following Scheme code:

```
(define x 42)
(define foo
  (lambda (y) (+ x y)))
(define bar
  (lambda (x) (foo x)))
(bar 2778)
```

(a) Assuming that Scheme has static scoping. (2 points)
Solution: 2820

(b) Assuming that Scheme has dynamic scoping. (2 points)
Solution: 5556

Now write the final output of the following Scheme code:

```
(define x 42)
(define foo
  ((lambda (y)
      (lambda (z) (+ y z))) x))
(define bar
  (lambda (x) (foo x)))
(bar 2778)
```

(c) Assuming that Scheme has static scoping. (3 points)
Solution: 2820

(d) Assuming that Scheme has dynamic scoping. (3 points)
Solution: 2820

```
(define foo
  (lambda (x y)
    ((lambda (z x) (+ x y z)) x)))
```

(e) Rewrite the above definition (define foo ...), replacing formal arguments (i.e., lambda variables) by their lexical addresses. (Note: Use 0-based indexing.) (5 points)

```
(define foo
  (lambda 2
    (lambda 2 (+ (: 0 1) (: 1 1) (: 0 0)) (: 0 0)))
```
4. Lambda Calculus (20 points)

A \textit{ground term} is an expression that can not be $\beta$-reduced any further.

(a) Perform $\alpha$-reduction on $(\lambda z. (\lambda y. (\lambda z. y \ z)) \ z)$, replacing $z$ with $x$. (5 points)

\textbf{Solution:} $(\lambda x. (\lambda y. (\lambda z. y \ z)) \ z)$

(b) Perform $\beta$-reduction on $((\lambda x. (\lambda y. x \ y)(\lambda x. x)) \ (\lambda a. a \ y \ x))$ (Note: Use lazy evaluation, or Call-By-Name. Only perform one $\beta$-reduction. You do not need to reduce the expression to a ground term.) (5 points)

\textbf{Solution:}

\[
((\lambda x. (\lambda w. x \ w)(\lambda x. x)) \ (\lambda a. a \ y \ x))
\]

\[
(\lambda w. ((\lambda a. a \ y \ x) \ w)(\lambda x. x))
\]

(c) Reduce $((\lambda x. \lambda y. x) ((\lambda x. \lambda y. y) \lambda a. b) ((\lambda x. x \ x) \ (\lambda x. x \ x)))$ to a ground term using Call-By-Value. If it is not possible to reduce the expression to a ground term, explain why. (5 points)

\textbf{Solution:} The expression can not be reduced to ground terms, because $(\lambda x. x \ x)(\lambda x. x \ x) \beta$-reduces to itself, so a $\beta$-reduction is always possible.

(d) Reduce the above using Call-By-Name ordering. If it is not possible to reduce the expression to a ground term, explain why. (5 points)

\textbf{Solution:}

\[
((\lambda x. \lambda y. x) ((\lambda x. \lambda y. y) \lambda a. b) ((\lambda x. x \ x) \ (\lambda x. x \ x)))
\]

\[
((\lambda y. ((\lambda x. \lambda y. y) \lambda a. b)) ((\lambda x. x \ x) \ (\lambda x. x \ x)))
\]

\[
((\lambda x. \lambda y. y) \lambda a. b)
\]

\[
(\lambda y. y)
\]
5. Regular Languages (20 points)

Consider the following NDFA:

(a) Give an equivalent regular expression. (10 points)

Solution: $0|(0(0|1)^*0)$

(b) Convert the above NDFA into a DFA. (10 points)
6. Grammars (20 points)

Consider the following Grammar:

\[
\begin{align*}
Exp & \rightarrow Exp + Exp \\
Exp & \rightarrow Exp - Exp \\
Exp & \rightarrow 0|1
\end{align*}
\]

(a) Show the parse tree for the following expressions. If multiple parses exist, show two different parses. (5 points)

i. \(1 + 0\)

\[
\begin{array}{c}
\text{Exp} \\
\downarrow \\
\text{Exp} + \text{Exp} \\
\downarrow \\
1 + 0
\end{array}
\]

ii. \(1 + 0 - 1\)

\[
\begin{array}{c}
\text{Exp} \\
\downarrow \\
\text{Exp} - \text{Exp} \\
\downarrow \\
\text{Exp} + \text{Exp} \\
\downarrow \\
\downarrow \\
1 + 0 \quad 1 + 0 - 1
\end{array}
\]

(b) Remove the ambiguity in the above grammar by making it left-associative. Write the modified grammar below. (15 points)

\[
\begin{align*}
Exp & \rightarrow Exp + Term \\
Exp & \rightarrow Exp - Term \\
Exp & \rightarrow Term \\
Term & \rightarrow 0|1
\end{align*}
\]