Problem Set 7
Due: 11:59pm Wednesday, May 4

Overview:
The individual problems on this assignment test your understanding of environment diagrams and translators. The group problems on this assignment cover closure conversion, imperative program, and parameter passing.

Reading:
- Handout #35: FOFL and FOBS: Restricted Functions
- Handout #37: Closure Conversion
- Handout #38: Imperative Programming
- Handout #39: Parameter Passing

Individual Problem Submission:
Each student should turn in a hardcopy submission packet for the individual problem by slipping it under Lyn’s office door by 11:59pm Wed. May 4. The packet should include:
1. an individual problem header sheet;
2. your environment diagram from Problem 1a.
3. your explanation for Problem 1b (including an environment diagram).
4. your final version of FoflPartialEval.ml from Problem 2.

Each student should also submit a softcopy (consisting of your final ps7-individual directory) to the drop directory by executing:

```
cd /students/username/cs251
cp -R ps7-individual ~cs251/drop/ps7/username/
```

Working Together:
If you want to work with a partner on this assignment, you should try to find a different partner than you worked with on a previous assignment. If this proves difficult, please email Lyn describing your situation.

Group Problem Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn’s office door by 11:59pm on Wed. May 4. The packet should include:
1. a team header sheet indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your pencil and paper solutions to Group Problems 1, 2, and 3a.
3. your final version of Counters.java from Problem 3b.
4. your final version of Process.java from Problem 4.

Each team should also submit a single softcopy (consisting of your final ps7-group directory) to the drop directory ~cs251/drop/p7/username, where username is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute:

```
cd /students/username/cs251
cp -R ps7-group ~cs251/drop/ps7/username/
```
Individual Problems

These are individual problems. Each student must solve this problem on her own without consulting any other person (except Lyn).

Individual Problem 1 [25]: Environmental Action

a. [20] Draw an environment diagram that shows all the environments and closures that are created when the following program is run on the input argument list [3;5] in statically scoped HoFL:

\[
\text{hofl (i j) (bind q (f p)) (bind r (q i j)) (bind answer (p (r m) (r (f m)))) answer))}
\]
\[
\text{(def (f g) (fun (a b) (g b a))) (def (p d e) (fun (h) (h d e))) (def (m x y) (% x y))}
\]

Follow the conventions for drawing environment diagrams used in Problem 1 of Problem Set 6. Think carefully about the parent pointer of each environment. Your diagram should accurately show the sharing of closure values. For example, if the same closure \( c \) is named \( I_1 \) in environment frame \( e_1 \) and \( I_2 \) in environment frame \( e_2 \), then \( c \) should appear exactly once in your diagram and there should be arrows from \( I_1 \) in \( e_1 \) to \( c \) and from \( I_2 \) in \( e_2 \) to \( c \).

b. [5] After carefully studying the description of HoFL and dynamic scope in Handout #34, Dina McScoop declares “The desugaring of multiple-argument function abstractions doesn’t work in dynamically scoped HoFL.” Explain what Dina means by showing that the desugaring of

\[
((\text{fun (a b) (+ a b)) 1 2})
\]

does not evaluate as desired in dynamically scoped HoFL. Draw an environment diagram as part of your explanation.

Individual Problem 2 [25]: Partial Evaluation

Avoiding Magic Constants

It is good programming style to avoid “magic constants” in code by explicitly calculating certain constants from others. For instance, consider the following two Fofl programs for converting years to seconds:

; Program 1
(fofl (years) (* 31536000 years))

; Program 2
(fofl (years) (bind seconds-per-minute 60 (bind minutes-per-hour 60 (bind hours-per-day 24 (bind days-per-year 365 ; ignore leap years (bind seconds-per-year (* seconds-per-minute (* minutes-per-hour (* hours-per-day days-per-year))) (* seconds-per-year years))))))))
The first program uses the magic constant 31536000, which is the number of seconds in a year.\(^1\) The second program shows how this constant is calculated from simpler constants. By showing the process by which `seconds-per-year` is calculated, the second program is a more robust and well-documented software artifact. Calculated constants also have the advantage that they are easier to modify. Although the numbers in the above program aren’t going to change, there are many so-called “constants” built into a program that change over its lifetime. For instance, the size of word of computer memory, the price of a first-class stamp, and the rate for a certain tax bracket are all numbers that could be hard-wired into programs but which might need to be updated in future version of the software.

However, magic constants can have performance advantages. In the above programs, the program with the magic constant performs one multiplication, while the other program performs four multiplications. If performance is critical, the programmer might avoid the clearer style and instead opt for magic constants.

**Partial Evaluation**

Is there a way to get the best of both approaches? Yes! We can write our program in the clearer style, and then automatically transform it to the more efficient style via a process known as partial evaluation. Partial evaluation transforms an input program into a residual program that has the same meaning by performing computation steps that would otherwise be performed when running the program. Any computation steps that can be performed during partial evaluation are steps that do not need to be performed when the residual program is run later. In most cases, the residual program has better run-time performance than the original program.

For instance, we can use partial evaluation to systematically derive the first program above from the second. We begin via a transformation known as constant propagation, in which we substitute the four constants at the top of the second program into their references to yield:

```
(fofl (years)
  (bind seconds-per-minute 60
   (bind minutes-per-hour 60
    (bind hours-per-day 24
     (bind days-per-year 365 ; ignore leap years
      (bind seconds-per-year (* 60 (* 60 (* 24 365)))
       (* seconds-per-year years)))))))
```

Next, we eliminate the now-unnecessary first four bindings via a transformation known as dead code removal:

```
(fofl (years)
  (bind seconds-per-year (* 60 (* 60 (* 24 365)))
   (* seconds-per-year years)))
```

We can now perform the three multiplications involving manifest integers in a step known as constant folding:

```
(fofl (years)
  (bind seconds-per-year 31536000
   (* seconds-per-year years)))
```

Finally, another round of constant propagation and dead code removal yields the first program:

\(^1\)It is worth noting that this number is approximately \(\pi \times 10^7\). So a century is approximately \(\pi \times 10^9\) seconds, which means that \(\pi\) seconds is approximately one nano-century!
It is not possible for partial evaluation to eliminate bindings whose definition ultimately depends on the program parameters. Nevertheless, it is often possible to partially simplify such definitions. For example, consider:

```
(fofl (a)
  (bind b (* 3 4)
    (bind c (+ a (- 15 b))
      (bind d (/ c b)
        (* d c))))
```

The transformation techniques described above can simplify this program to:

```
(fofl (a)
  (bind c (+ a 3)
    (bind d (/ c 12)
      (* d c))))
```

In this example, (+ a (- 15 b)) cannot be replaced by a number (because the value of a is unknown), but it can be simplified to the residual expression (+ a 3). Similarly, (/ c b) is transformed to the residual expression (/ c 12) and (bind b ...) is transformed to the residual expression

```
(bind c (+ a 3)
  (bind d (/ c 12)
    (* d c)))
```

Your Task

In this problem, your task is to write a function `partialEval` that performs partial evaluation on a Fofl program. Given a Fofl program, `partialEval` should return another Fofl program that has the same meaning as the original program, but which also satisfies the following properties:

1. The resulting program should not contain any `bind` expressions in which a variable is bound to a literal value.
2. The resulting program should not contain any primitive applications in which a primitive operator is applied to operands that are all literal values. An important exception to this property is the case where performing the primitive operator would result in a run-time error. In this case the primitive application should be left as a residual expression. For example, (* 3 0), (% 5 0), (+ 1 #t), and (and 2 #f) are examples of primitive applications with literal operands that should remain as residual expressions.
3. The resulting program should not contain any conditionals in which the test expression is #t or #f. Such expressions should be replaced by the appropriate branch.

It is possible to write separate functions that perform the constant propagation, constant folding, and dead-code elimination steps, but it is tricky to get them to work together to perform all transformations. It turns out that it is much more straightforward to perform all three kinds of transformations at the same time in a single walk over the expression tree. This can be accomplished with a pair of functions:

```
val partialEval: Fofl.pgm -> Fofl.pgm
  Returns a partially evaluated version of the given Fofl program.
```
val peval: Fofl.exp -> Fofl.valu Env.env -> Fofl.exp

Given a Fofl expression exp and an partial evaluation environment env, returns the simplified version of exp. The partial evaluation environment contains name/value bindings for names whose values are known. In general, only some of the names in an expression will have known values.

Your goal is to implement simplification by fleshing out these two function definitions in the file ~/ps7-individual/FoflPartialEval.ml.

Note that there is a correspondence between run/eval in FoflEnvInterp and partialEval/peval. peval is effectively a version of eval that evaluates as much of an expression as it can based on the “partial” environment information it is given. Because bindings for some names may be missing in the environment, peval cannot always evaluate every expression to the value it denotes and in some cases must instead return a residual expression that will determine the value when the program is executed. Because of this, peval must always return an expression rather than a value; even in the case where it can determine the value of an expression, that value must be expressed as a literal expression of type exp, not a value of type valu.

Notes:

- Perform #use "load-peval.ml" to load the partial evaluator.
- In some cases it would be possible to perform more aggressive transformations if you took advantage of algebraic properties like the associativity and commutativity of addition and multiplication. To simplify this problem, you should not use any algebraic properties of the arithmetic operators. For example, you should not transform (+ 1 (+ a 2)) into (+ 3 a), but should leave it as is. You should not even perform “obvious” transformations like (+ 0 a) ⇒ a, (* 1 a) ⇒ a, and (* 0 a) ⇒ 0. Although the first two of these transformations are valid, the last is unsafe in the sense that it can change the meaning of a program. For instance, (* 0 (/ a b)) cannot be simplified to 0, because it does not preserve the meaning of the program in the case where b is 0 (in which case evaluating the expression should give an error).
- Your partial evaluator should not try to perform any applications of user-defined functions. For example, it should transform the program

(fofl () (avg (sq (+ 1 2)) (+ 3 4))
  (def (sq x) (* x x))
  (def (avg a b) (bind n (+ 1 1) (/ (+ a b) n))))

to

(fofl () (avg (sq 3) 7)
  (def (sq x) (* x x))
  (def (avg a b) (/ (+ a b) 2))

In this example, note that the operand expressions of function applications have been simplified, but the function applications themselves have been left as residual expressions. Also note that the bodies of function definitions have been simplified.

Why shouldn’t the partial evaluator perform function applications when all the arguments are known? Because we want the partial evaluator to be guaranteed to terminate. Fofl function declarations may be recursive in general, and applying a recursive function is not guaranteed to terminate. It is possible to analyze the function definitions to conservatively approximate which functions are guaranteed to terminate and which functions might not terminate, but such an analysis is beyond the scope of this problem.
The following auxiliary functions have been provided for you in FoflPartialEval:

```ocaml
define isLit (exp) =
    match exp with
    | Lit _ -> true
    | _    -> false

define litVal (exp) =
    match exp with
    | Lit v -> v
    | _     -> raise (Failure ("not a lit" ^ (expToString exp)))
```

Your partial evaluator should be concise. If you find yourself writing a lot of code, you are on the wrong track. Except for the two auxiliary functions provided for you (isLit and litVal), you should not need to define any functions other than partialEval and peval. (But you may define other auxiliary functions if you desire.)

To handle the special case in which applying a primitive operator to literal operands yields an error, use the following OCAML exception handling idiom:

```ocaml
try <expression for normal case>
with EvalError e -> <expression for exceptional case>
```

You can use the testPgm function (which takes a string representation of a FOFL program) to test your partial evaluator. For example:

```ocaml
# testPgm "(fofl () (+ 1 2))";;
(fofl () 3)
- : unit = ()

# testPgm "(fofl (a b) (bind c (+ 1 2) (if (= c 4) (+ a b) (* a b))))";;
(fofl (a b) (* a b))
- : unit = ()

# testPgm "(fofl (a)
        (+ (* (+ 1 2) a)
         (+ (* 3 4)
          (+ (* 0 a)
           (+ (* 1 a)
            (+ 0 a))))))";;
(fofl (a) (+ (* 3 a) (+ 12 (* (* 0 a) (+ (* 0 a) (+ 0 a)))))
- : unit = ()

# testPgm "(fofl () (+ (* 2 3)
        (+ (/ 4 0)
         (+ (% 5 0)
          (+ (and 6 #t)
           (+ 7 #f))))))";;
(fofl () (+ 6 (+ (/ 4 0) (+ (% 5 0) (+ (and 6 #t) (+ 7 #f))))))
- : unit = ()
```

You can also test your partial evaluator by evaluating test(). This applies your partial evaluator to all the test entries in the list testEntries in the file FoflPartialEvalTest.ml. The entries in this list are by no means exhaustive. You are strongly encouraged to add more entries to this list.
Group Problems

Group Problem 1 [16]: Safe Transformations

A transformation that rewrites one expression to another is said to be safe if performing the transformation anywhere in a program will not change the behavior of the program. For each of the following transformations, indicate whether it is safe in (i) HOFL and (ii) HOILEC. For each transformation you specify as unsafe, give an example whose behavior is changed by the transformation. Changes in behavior include:

- the program returns different values before and after the transformation.
- the program loops infinitely before the transformation, but returns a value after the transformation.
- the program returns a value before the transformation, but loops infinitely after the transformation.

In each expression, I stands for a variable reference and E stands for an expression. You may assume that all subexpressions of an application are evaluated in left-to-right order.

a. \( (+ \ I \ I) \Rightarrow (\ast \ 2 \ I) \)
b. \( (+ \ E \ E) \Rightarrow (\ast \ 2 \ E) \)
c. \( (+ \ E_1 \ E_2) \Rightarrow (+ \ E_2 \ E_1) \)
d. \( (+ \ E_1 \ E_2) \Rightarrow (\text{bind} \ x \ E_1 \ (+ \ x \ E_2)) \)
e. \( (+ \ E_1 \ E_2) \Rightarrow (\text{bindpar} \ ((x \ (\text{fun} () \ E_1)))
                           (y \ (\text{fun} () \ E_2)))
                           (+ \ (x) \ (y))) \)
f. \( (\text{if} \ #t \ E_1 \ E_2) \Rightarrow E_1 \)
g. \( (\text{if} \ E_1 \ E_2 \ E_2) \Rightarrow E_2 \)
h. \( (\text{if} \ (\text{if} \ E_1 \ E_2 \ E_3) \ E_4 \ E_5) \Rightarrow (\text{if} \ E_1 \ (\text{if} \ E_2 \ E_4 \ E_5) \ (\text{if} \ E_3 \ E_4 \ E_5)) \)

Group Problem 2 [24]: Parameter Passing

Consider the following HOILIC expression:

\[
\begin{align*}
(bind \ a \ 1 \\
(bind \ inc! \ (\text{fun} () \ (\text{seq} \ (<- \ a \ (+ \ a \ 1))) \ a)) \\
(bind \ f \ (\text{fun} \ (y \ z) \\
(\text{seq} \ (<- \ y \ (+ \ y \ 3)) \\
(\ast \ a \ (* \ z \ z)))) \\
(f \ a \ (inc!)))
\end{align*}
\]

For each of the following parameter-pasing mechanisms, (i) draw a diagram that shows how the above expression is evaluated in lexically-scoped HOILIC using that parameter-passing mechanism and (ii) indicate the value of the expression. You should assume that all operands are evaluated in left-to-right order.

- Call-by-value
• Call-by-reference
• Call-by-name
• Call-by-lazy (i.e., call-by-need)

Note: Your ~/cs251/hoilic directory contains interpreters for all four parameter passing mechanisms. You can use them to check your answers, but the diagrams are essential.

Group Problem 3 [25]: Counters
Recall that in HOILIC (1) every variable name is bound to an implicit cell; (2) references to a variable implicitly dereference (return the contents of) the cell; and (3) a variable $v$ can be assigned a new value via the assignment construct ($v <- E$), which changes the contents of the implicit cell associated with $v$ to the value of $E$.

a. [15] Consider the functions in Fig. 1, which are written in call-by-value lexically-scoped HOILIC. For each of the following expressions, (1) give the value of the expression and (2) draw an environment diagram that justifies why the expression has that value. You should assume that all operands are evaluated in left-to-right order.

• (test-counter make-counter1)
• (test-counter make-counter2)
• (test-counter make-counter3)

b. [10] Let $i$ range over the numbers $\{1, 2, 3\}$. Then each of the HOILIC functions make-counter$i$ can be modeled in JAVA by an instance of class Counter$i$ that implements the following interface:

```java
interface Counter {
    public int invoke();
}
```

In addition to its single nullary instance method invoke, each class Counter$i$ should have a single class, instance, or local variable named count. The test expression (test-counter make-counter$i$) in HOILIC can be modeled by the JAVA statement:

```java
Counters.testCounters(new Counter1(), new Counter1());
```

where testCounters is a class method of the Counters class with the following definition:

```java
public static void testCounters (Counter a, Counter b) {
    return IL.prepend(a.invoke(),
                      IL.prepend(b.invoke(),
                                IL.prepend(a.invoke(),
                                          IL.empty())));
}
```

Here IL. is a prefix for operations manipulating integer lists.

In this subproblem your task is to flesh out the definitions of the Counter$i$ classes in the file Counters.java so that they correctly model make-counter$i$.

Notes:
• Appendix A presents an API for the Java IntList class. An implementation of this class (which you do not need to study) is provided in ~/cs251/ps7-group/IntList.java.
• To compile and run the Counters program, execute the following Linux shell commands:
  cd "~/cs251/ps7-group"
  javac Counters
  java Counters

(def make-counter1
  (bind count 0
    (fun ()
      (fun ()
        (seq (<- count (+ count 1))
          count))))))

(def make-counter2
  (fun ()
    (bind count 0
      (fun ()
        (seq (<- count (+ count 1))
          count))))))

(def make-counter3
  (fun ()
    (fun ()
      (bind count 0
        (seq (<- count (+ count 1))
          count))))))

(def test-counter
  (fun (make-counter)
    (bindseq ((a (make-counter))
      (b (make-counter)))
      (list a) (b) (a))))

Figure 1: HOILIC counter functions.

Group Problem 4 [35]: Manually Converting OCAML to JAVA

In this problem you will manually translate an OCAML program with block structure and higher-order functions into a JAVA program that has neither block structure nor higher-order functions.

Consider the OCAML list-processing function process in Fig. 2. Given an input integer list, process generates an output integer list. You should study the definition of process carefully to understand what it does. Here are some examples of process in action:

process [3;4;5;6;7];;
  - : int list = [1; 2; 3; 13; 15; 17; 19; 21]

# process [5;7;4;5;7];;
  - : int list = [2; 3; 2; 3; 35; 47; 29; 35; 47]

# process [5;7;4;6;5;7];;
  - : int list = [2; 3; 2; 3; 15; 17; 14; 16; 15; 17]

# process [5;7;4;6;8;5;7];;
  - : int list = [2; 3; 2; 3; 35; 47; 29; 41; 53; 35; 47]
let process xs =
let rec scan1 ys f =
  match ys with
  [] -> mapxs f
| (y::ys') ->
    if (y mod 2) = 0 then
      scan2 ys' f
    else
      let y' = y / 2
      in y' :: (scan1 ys' (fun a -> f (a + y')))
and scan2 zs g =
match zs with
  [] -> mapxs g
| (z::zs') ->
    if (z mod 2) = 0 then
      scan1 zs' g
    else
      let z' = z / 2
      in z' :: (scan2 zs' (fun b -> g (b * z')))
and mapxs q =
  let rec mapq ws =
    match ws with
    [] -> []
  | (w::ws') -> (q w)::(mapq ws')
in mapq xs
in scan1 xs (fun x -> x)

Figure 2: The OCAML process function.

# process [1;2;3];;
- : int list = [0; 1; 1; 2; 3]
# process [3;2;1];;
- : int list = [1; 0; 1; 1; 1]
# process [1;3;5];;
- : int list = [0; 1; 2; 4; 6; 8]
# process [2;4;6];;
- : int list = [2; 4; 6]

Your task in this problem is to translate the OCAML process function and its internal functions into JAVA methods that perform the same computations. You should do this by filling out the skeleton file Process.java (Fig. 3) that can be found in the ps7-group directory. In the Process class, you should write methods process, scan1, scan2, mapxs, and mapq that correspond to the five list-processing functions with the same names in Fig. 2. These functions may need to take additional arguments due to the “flattening” of the OCAML block structure that must be done as part of translating the functions into JAVA. The higher-order functions (i.e., closure values) occurring in the OCAML program can be translated into instances of classes that implement the IntFun interface in Process.ml. One such class, IdFun, which implements identity functions, has been provided for you. You will have to define other classes implementing this interface, either explicitly (via JAVA class definitions) or implicitly (using anonymous inner classes).
Notes:

- The OCAML process function can be found in the file `~/cs251-ps7-group/process.ml`.
- Appendix A presents an API for the Java IntList class. An implementation of this class (which you do not need to study) is provided in `~/cs251-ps7-group/IntList.java`.
- Use `javac` to compile your file and `java` to test it. E.g.:
  
  ```
  [gdome@jaguar gdome] cd ~/cs251-ps7-group
  [gdome@jaguar ps7-group] javac Process.java
  [gdome@jaguar ps7-group] java Process [3,4,5,6,7]
  [1,2,3,13,15,17,19,21]
  [gdome@jaguar ps7-group] java Process [5,7,4,5,7]
  [2,3,2,3,35,47,29,35,47]
  [gdome@jaguar ps7-group] java Process [1,2,3]
  [0,1,1,2,3]
  [gdome@jaguar ps7-group] java Process [2,4,6]
  [2,4,6]
  [gdome@jaguar ps7-group]
  ```
// Interface for int -> int functions
interface IntFun {
    public int apply (int x);
}

// The identity function
class IdFun implements IntFun {
    public int apply (int x) {
        return x;
    }
}

// Put other classes implementing the IntFun interface here:
// (Alternatively, you can use anonymous inner classes instead.)

// The Process class defines the process, scan1, scan2, mapxs and mapq methods.
public class Process {

    // Handy way of introducing abbreviation IL. for IntList operations:
    // IL.empty, IL.isEmpty, IL.head, IL.tail, IL.prepend.
    public static IntList IL;

    // Define process here:
    // Define scan1 here:
    // Define scan2 here:
    // Define mapxs here:
    // Define mapq here:

    // Testing method. E.g.:
    // [lyn@jaguar ps7-group] java Process [3,4,5,6,7]
    // [1,2,3,13,17,19,21]
    // [lyn@jaguar ps7-group] java Process [5,7,4,5,7]
    // [2,3,2,3,35,47,29,35,47]
    public static void main (String [] args) {
        if (args.length == 1) {
            System.out.println(process(IL.fromString(args[0])));
        } else {
            System.out.println("unrecognized main option");
        }
    }
}

Figure 3: The JAVA file Process.java.
Appendix A: An IntList Class for Java

Problems 3 and 4 use a Java IntList class that provides list manipulation functions on immutable integer lists. It has the API given below. The file ~/cs251/ps7-group/IntList.java contains an implementation of this class.

```java
public static IntList prepend (int i, IntList L);
// Returns the new list that results from prepending i onto L.

public static IntList empty ();
// Returns an empty list.

public static boolean isEmpty (IntList L );
// Returns true if L is empty and false otherwise.

public static int head (IntList L);
// Returns the head of list L. Throws an exception if L is empty.

public static IntList tail (IntList L);
// Returns the tail of list L. Throws an exception if L is empty.

public String toString ();
// Returns a string representation of this list.

public static String toString (IntList L);
// Returns a string representation of list L. This is a sequence
// of comma-separated integers delimited by square brackets.

public static IntList fromString (String s);
// Returns the integer list whose string representation is s.
```

If a Java program contains the following declaration, then The abbreviation IL. may be used in place of IntList.; For example, IL.head rather than IntList.head:

```java
static IntList IL;
```
Name:

Date & Time Submitted:

By signing below, I attest that I have followed the policy for individual problems set forth in the Course Information handout. In particular, I have not consulted with any person except Lyn about these problems and I have not consulted any materials from previous semesters of CS251.

Signature:

In the Time column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading you problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1a [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1b [5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CS251 Problem Set 7 Group Problems
Due 11:59pm Wednesday, May 4

Names of Team Members:

Date & Time Submitted:

Collaborators (anyone you or your team collaborated with):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

In the Time column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading you problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1 [16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [24]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4 [35]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>