CS251 Programming Languages **Handout # 24**
Prof. Lyn Turbak **February 25, 2005**
Wellesley College

Problem Set 4
Due: 6pm Friday, March 2

Overview:

The individual problem on this assignment tests your understanding of highe-order list operations. The group problems on this assignment will give you practice with the signal-processing style of programming, modules, and sum-of-product datatypes.

Reading:

- Handout #8 (Jason Hickey's OCAML tutorial): Chapters 6, 7, 10 (ignore 10.4), and 11.
- Handout #22: John Backus's Turing Award Lecture (Sections 1 – 11 and 15–16)
- Handout #23: Modules and Data Abstraction in OCAML.
- Handout #25: Sum-of-Product Data Types

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 6pm Fri. March. 2. The packet should include:

1. an individual problem header sheet;
2. your final version of `disjoint.ml` from Problem 1;
3. a transcript showing the result of running `test_disjoint()`;

Each student should also submit a softcopy (consisting of your final `ps4-individual` directory) to the drop directory `~cs251/drop/ps4/`*username*.

Working Together:

If you want to work with a partner on this assignment, try to find a different partner than you worked with on a previous assignment. If this is not possible, you may choose a partner from before. But try not to choose the same partner you chose last week!

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 6m on Fri. Mar. 2. 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 answers for Group Problem 1;
3. your final version of `PredSet.ml` for Group Problem 2;
4. a transcript of test cases showing that your `PredSet` functions work;
5. your final version of `OperationTreeSet.ml` for Group Problem 3;

Each team should also submit a single softcopy (consisting of your final `ps4-group` directory) to the drop directory `~cs251/drop/p4/`*username*, where *username* is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet).

Individual Problem [20]: Feeling Disjointed

Consider the following OCaml list function:

val `pairwise_disjoint: 'a list list -> bool`

`pairwise_disjoint xss` returns `true` if no two lists in `xss` share an element in common. Otherwise (i.e., there are two lists in `xss` that share an element in common) it returns `false`.

For example:

```
# pairwise_disjoint [];;
- : bool = true

# pairwise_disjoint [[1]];;
- : bool = true

# pairwise_disjoint [[1];[2]];;
- : bool = true

# pairwise_disjoint [[1;1];[2;2]];; (* There may be duplicates within a list *)
- : bool = true

# pairwise_disjoint [[1;2];[1]];;
- : bool = false

# pairwise_disjoint [[1;2];[2]];;
- : bool = false

# pairwise_disjoint [[1;2];[3;4;5];[6;7;8;9]];;
- : bool = true

# pairwise_disjoint [[1;2;1];[3;4;3;5];[6;7;6;8;8;9]];; (* Dups within a list OK *)
- : bool = true

# pairwise_disjoint [[6;1;4];[3;9];[8;2;7;5]];;
- : bool = true

# pairwise_disjoint [[6;1;7;4];[3;9];[8;2;7;5]];;
- : bool = false

# pairwise_disjoint [[6;1;4];[3;9;1];[8;2;7;5]];;
- : bool = false

# pairwise_disjoint [[6;1;4];[3;9];[8;3;2;7;5]];;
- : bool = false

# pairwise_disjoint [[5;45;25;35;15];[1];[3;23;13];[34;14;24;4];[2;12]];;
- : bool = true

# pairwise_disjoint [[5;45;2;35;15];[1];[3;23;13];[34;14;24;4];[2;12]];;
- : bool = false

# pairwise_disjoint [[5;45;25;35;15];[1];[3;23;13];[34;12;24;4];[2;12]];;
- : bool = false

# pairwise_disjoint [[5;45;25;35;15];[14];[3;23;13];[34;14;24;4];[2;12]];;
- : bool = false
```

Your task in this problem is to flesh out a definition for the `pairwise_disjoint` function, which can be found in the file `~/cs251/ps4-individual/disjoint.ml`. Your definition should *not* use any explict recursion but should instead use the higher-order list functions in the `ListUtils` module (which can be found in `~/cs251/utils/ListUtils.ml`).

Notes:

- Use `#use "load-disjoint.ml"` to load all files relevant to the problem. This will load `FunUtils.ml`, `ListUtils.ml`, and some testing code in addition to your definition(s) from `disjoint.ml`.
- The file `disjoint.ml` begins with the declarations:

  ```
  open FunUtils
  open ListUtils
  ```

  This makes all functions in the `FunUtils` and `ListUtils` modules available in `funs.ml` without the need for explicit qualification. E.g., you can write `id` rather than `FunUtils.id` and `map` rather than `ListUtils.map`.
- You *may* define any auxiliary functions you find helpful, but these should not use explicit recursion either.
- If you cannot think of any way to solve some part of the problem except by using recursion, you may use recursion for partial credit.
- Use `test_disjoint()` to test your `pairwise_disjoint` function on a sample testing suite. After changing your definition of `pairwise_disjoint`, you should reload all files via

  ```
  #use "load-disjoint.ml"
  ```

  before invoking `test_disjoint()`.

Group Problems

Group Problem 1 [35]: Backus's Paper

This problem is about John Backus's 1977 Turing Award Lecture: *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.* You should begin this problem by reading Sections 1 – 11 and 15–16 of this paper. (Although Sections 12–14 are very interesting, they require more time than I want you to spend on this problem.)

Section 11.2 introduces the details of the FP language. Backus uses many notations that may be unfamiliar to you. For example,

- $p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n; e_{n+1}$ is similar to the following OCAML expression:

```
if p_1 then e_1
else ...
else if p_n then p_n
else e_{n+1}
```

- $\langle e_1, \ldots, e_n \rangle$ denotes the sequence of the n values of the expressions e_1, …, e_n. ϕ denotes the empty sequence. Because FP is dynamically typed, such sequences can represent both tuples and lists from OCAML.
- The symbol $\perp$ (pronounced "bottom") denotes the value of an expression that doesn't terminate (i.e., it loops infinitely) or terminates with an error.

Consult Lyn if you have trouble understanding Backus's notation.

a. **[10]** Write a few paragraphs summarizing the "big ideas" in the sections of the paper that you have been assigned.

b. **[5]** One of the reasons this paper is well-known is that in it Backus coined the term "von Neumann bottleneck". Describe what this is and its relevance to the paper.

c. **[5]** The FP language Backus introduces in Section 11 does not support abstraction expressions like OCAML's `fun` and SCHEME's `lambda`. Why did Backus make this decision in FP?

d. **[15]** Consider the following FP definition:

$$\textbf{Def } F \equiv \alpha/+ \circ \alpha\alpha\times \circ \alpha\text{distl} \circ \text{distr} \circ [\text{id}, \text{id}]$$

What is the value of $F\langle 2, 3, 5\rangle$? Show the evaluation of this expression in algebra-like steps.

Group Problem 2 [30]: Functional Sets

In OCAML, we can implement abstract data types in terms of familiar structures like lists, arrays, and trees. But we can also use functions to implement data types. Here we show a compelling example of using functions to implement sets. Rather than using the `SET` signature used in Handout #23 (see Fig. 1)[1] we will use the somewhat different `PRED_SET` signature shown in Fig. 2. Here is a comparison of `PRED_SET` with `SET`:

- `PRED_SET` has many of the same operations as `SET`: `empty`, `singleton`, `member`, `union`, `intersection`, `difference`, and `fromList`

[1]The `SET` signature in Fig. 1 includes two functions, `fromSexp` and `toSexp` that were not mentioned in Handout #23, but are actually part of the set implementation you will use in Group Problem 3.

- PRED_SET does *not* support the following operations of SET: isEmpty, size, toList, fromSexp, toSexp, or toString.
- PRED_SET has two operations that SET does not have: fromPred and toPred. These allow converting between predicates and sets.

```
module type SET = sig
  type 'a set
  val empty : 'a set                    (* the empty set *)
  val singleton : 'a -> 'a set          (* a set with one element *)
  val insert : 'a -> 'a set -> 'a set   (* insert elt into given set *)
  val delete : 'a -> 'a set -> 'a set   (* delete elt from given set *)
  val isEmpty: 'a set -> bool           (* is the given set empty? *)
  val size : 'a set -> int              (* number of distinct elements in given set *)
  val member : 'a -> 'a set -> bool     (* is elt a member of given set? *)
  val union: 'a set -> 'a set -> 'a set (* union of two sets *)
  val intersection: 'a set -> 'a set -> 'a set (* interscetion of two sets *)
  val difference: 'a set -> 'a set -> 'a set   (* difference of two sets *)
  val fromList : 'a list -> 'a set      (* create a set from a list *)
  val toList : 'a set -> 'a list        (* list all set elts, sorted low to high *)
  val fromSexp : (Sexp.sexp -> 'a)
                -> Sexp.sexp -> 'a set (* translates s-expression rep. into set *)
  val toSexp : ('a -> Sexp.sexp)
               -> 'a set -> Sexp.sexp  (* translates set into s-expression rep. *)
  val toString : ('a -> string)
                -> 'a set -> string    (* string representation of the set *)
end
```

Figure 1: The SET signature.

```
module type PRED_SET = sig
  type 'a set
  val empty: 'a set
  val singleton: 'a -> 'a set
  val member: 'a -> 'a set -> bool
  val union: 'a set -> 'a set -> 'a set
  val intersection: 'a set -> 'a set -> 'a set
  val difference:'a set -> 'a set -> 'a set
  val fromList: 'a list -> 'a set
  val fromPred: ('a -> bool) -> 'a set
  val toPred: 'a set -> ('a -> bool)
end
```

Figure 2: A signature for a version of sets based upon predicates.

The `fromPred` and `toPred` operations are based on the observation that a membership predicate describes exactly which elements are in the set and which are not. Consider the following example:

```
# let s = fromPred (fun x -> (x = 2) || (x = 3) || (x = 5));;
val s : int PredSet.set = <abstr>
# member 3 s;;
- : bool = true
# member 5 s;;
- : bool = true
# member 4 s;;
- : bool = false
# member 100 s;;
- : bool = false
```

The set `s` consists of exactly those elements satisfying the predicate passed to `fromPred` – in this case, the integers 2, 3, and 5.

Defining sets in terms of predicates has many benefits. Most important, it is easy to specify sets that have infinite numbers of elements! For example, the set of all even integers can be expressed as:

```
fromPred (fun x -> (x mod 2) = 0)
```

This predicate is true of even integers, but is false for all other integers. The set of all values of a given type is expressed as `fromPred (fun x -> true)`. Many large finite sets are also easy to specify. For example, the set of all integers between 251 and 6821 (inclusive) can be expressed as:

```
fromPred (fun x -> (x >= 251) && (x <= 6821))
```

a. [20]: `PredSet`

The most obvious way to implement the `PRED_SET` signature is in a module `PredSet` that defines the `set` type as a predicate:

```
type 'a set = 'a -> bool
```

Based on this representation, flesh out all the function definitions in the the `PredSet` module in the file `~/cs251/ps4-group/PredSet.ml`. Each of your definitions should be a one-liner. For example, the definition of `member` is

```
let member x s = s x
```

In the predicate representation, you can always write a set as

`fun y ->` *`expression determining if y is in the set`*

For example, you can write the `union` definition as

`let union s1 s2 = fun y ->` *`expression determining if y is in the union of s1 and s2`*

Most other function definitions in `PredSet` can be expressed in a similar way. In `fromList` (which can also be written in this style), you may use operations from the `List` or `ListUtils` modules.

Use `#use "PredSet.ml"` to load your module and `open PredSet` to make names in the module accessible without qualification. Convince yourself that your implementation is correct by making some simple sets and testing various set operations on them. Your hardcopy submission for this problem should include a transcript of your test cases.

b. [10]: Other Set Functions

In this part, you are asked to consider whether it is possible to implement the `SET` and `PRED_SET` signatures if we extend them with additional functionality. Explain all your answers.

1. Can we add to the `PRED_SET` signature the following function?

   `val toList: 'a set -> 'a list`
   Returns a list of all the elements in set.

2. Can we add to the `SET` signature the following function?

   `val fromPred: ('a -> bool) -> 'a set`
   Returns a set of all elements satsifying the given predicate.

3. Can we add the following function to the `SET` signature? To the `PRED_SET` signature?

   `val isEmpty: 'a set -> bool`
   Returns `true` if the set is empty, and `false` otherwise.

4. Can we add the following function to the `SET` signature? To the `PRED_SET` signature?

   `val complement: 'a set -> 'a set`
   Returns the complement of the given set – i.e., all the value of type `'a` that are not in the given set.

5. Can we add the following function to the `SET` signature? To the `PRED_SET` signature?

   `val isSubset: 'a set -> 'a set -> bool`
   Returns `true` if all of the elements of the first set parameter are are also elements of the second set parameter, and `false` otherwise.

Group Problem 3 [35]: `OperationTreeSet`

Background

In this problem, you will flesh out an implementation of the `SET` signature in Fig. 1. This is the same as the `SET` signature presented in Handout #23 *except* that it includes a `toSexp` that translates a set into an s-expression representation and a `fromSexp` function that translates an s-expression representation into a set. Different set implementations may have different s-expression representations. However, it should be the case that for all sets `s`, `fromSexp (toSexp s)` yields a set with exactly the same elements as `s`.

Before beginning this problem, you should study the sorted set implementation of sets and the BST implementation of sets:

- The sorted set implementation of sets is described in Section 4.2 of the *Modules* handout (#23). The code for this implementation is in `~/cs251/sets/SortedListSet.ml`. You can test it via the following OCAML commands:

  ```
  #cd "/students/username/cs251/sets";;
  #use "load-sorted-list-set.ml";;
  testZZZ();;
  ```

  where *ZZZ* is one of `Tiny`, `Small`, `Medium`, or `Large`. Each `testZZZ` function performs tests on word files of different sizes: the tiny file has 16 words, the small file has 476 words, the medium file has 5525 words, and the large file has 45425 words. (Because this is an inefficient representation, `testLarge()` takes a very long time to execute.)

- The BST implementation of sets is described in Section 2.3 of the *Sum of Products* handout (#25). The code for this implementation is in `~/cs251/sets/BSTSet.ml`. You can test it via the following OCAML commands:

  ```
  #cd "/students/username/cs251/sets";;
  #use "load-bst-set.ml";;
  testZZZ();;
  ```

where *ZZZ* is one of `Tiny`, `Small`, `Medium`, or `Large`.

Important: It turns out that some of the above testing functions (in particular, `testLarge()`) require more stack space than is provided by OCAML by default. In order to declare that OCAML should have more stack space, you need to perform the following steps exactly once (and everything should be set after that):

1. Add the following line to the end of your `~/.bashrc` file:

   ```
   export OCAMLRUNPARAM='l=10M'
   ```

   Note that the character `'l'` is a lowercase `'L'` and not the digit `'1'`. This tells OCAML to allocate 10 megawords of stack space (40 times greater than the default 250 kilowords).

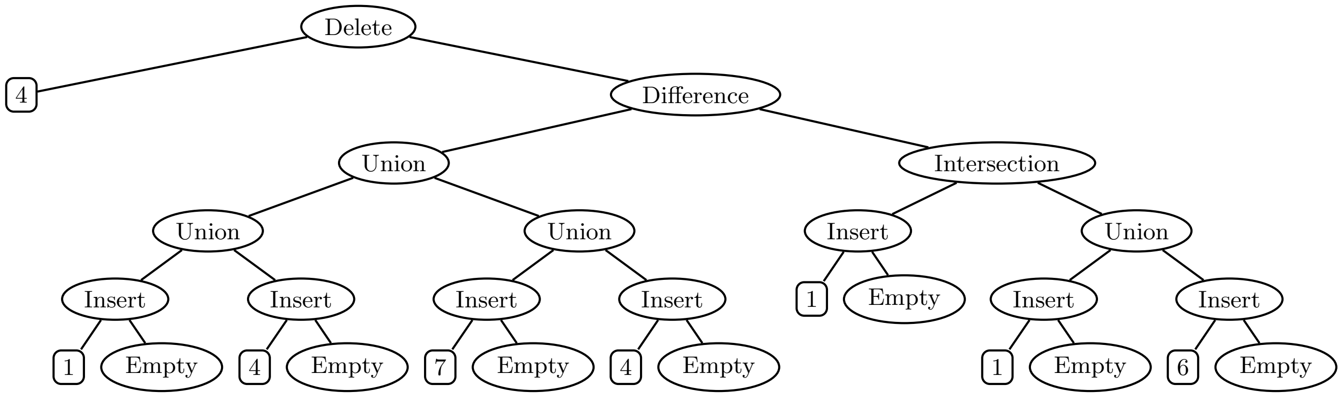
2. After modifying your `~/.bashrc` file, log out of Linux and then log back in. You should now be all set.

Operation Tree Representation of Sets

A very different way of representing a set as a tree is to remember the structure of the set operations `empty`, `insert`, `delete`, `union`, `intersection`, and `difference` used to create the set. For example, consider the set `t` created as follows:

```
let t = (delete 4 (difference (union (union (insert 1 empty)
                                            (insert 4 empty))
                                     (union (insert 7 empty)
                                            (insert 4 empty)))
                              (intersection (insert 1 empty)
                                            (union (insert 1 empty)
                                                   (insert 6 empty)))))
```

Abstractly, `t` is the singleton set {7}. But one concrete representation of `t` is the following operation tree:



One advantage of using such operation trees to represent sets is that the `empty`, `insert`, `delete`, `union`, `difference`, and `intersection` operations are *extremely* cheap – they just create a new tree node with the operands as subtrees, and thus take constant time and space! But other operations, such as `member` and `toList`, can be more expensive than in other implementations.

Your Task

In this problem, you are asked to flesh out the missing operations in the skeleton of the `OperationTreeSet` module (Fig. 3) in the file `~/cs251/ps4-group/OperationTreeSet.ml`. In this module, the `set` datatype is create by constructors `Empty`, `Insert`, `Delete`, `Union`, `Intersection`, and `Difference`. The `empty`, `singleton`, `insert`, `delete`, `union`, `intersection`, `difference`, and `toString` operations are trival and have already been implemented. You are responsible for

```
module OperationTreeSet : SET = struct

  module LSU = ListSetUtils

  type 'a set =
      Empty
    | Insert of 'a * 'a set
    | Delete of 'a * 'a set
    | Union  of 'a set * 'a set
    | Intersection of 'a set * 'a set
    | Difference of 'a set * 'a set

  let empty = Empty
  let insert x s = Insert(x,s)
  let singleton x = Insert(x, Empty)
  let delete x s = Delete(x, s)
  let union s1 s2 = Union(s1,s2)
  let intersection s1 s2 = Intersection(s1,s2)
  let difference s1 s2 = Difference(s1,s2)

  let rec toList s = (* Replace this stub. You may use operations in ListSetUtils,
                        using the abbreviation LSU defined above. *)
    match s with
      Empty -> []
    | Insert(y,s') -> []
    | Delete(y,s') -> []
    | Union(s1,s2) -> []
    | Intersection(s1,s2) -> []
    | Difference(s1,s2) -> []

  let rec fromList xs = Empty (* Replace this stub. You should define this in terms of
                                 a "balanced" tree of Union, Insert, and Empty nodes *)

  let rec member x s = (* Replace this stub. Do *not* use toList in this definition! *)
    match s with
      Empty -> true
    | Insert(y,s') -> true
    | Delete(y,s') -> true
    | Union(s1,s2) -> true
    | Intersection(s1,s2) -> true
    | Difference(s1,s2) -> true

  let size s = 17 (* Replace this stub. You *may* use toList in this definition. *)

  let isEmpty s = false (* Replace this stub. You *may* use toList in this definition. *)

  let rec toSexp eltToSexp s = (* Replace this stub. This function returns an
                                  s-expression that shows the structure of the tree.
                                  See the PS4 Problem 3 description for examples *)
    match s with
      Empty -> Sexp.Seq []
    | Insert(y,s') -> Sexp.Seq []
    | Delete(y,s') -> Sexp.Seq []
    | Union(s1,s2) -> Sexp.Seq []
    | Difference(s1,s2) -> Sexp.Seq []
    | Intersection(s1,s2) -> Sexp.Seq []

  let rec fromSexp eltFromSexp sexp = Empty (* Replace this stub *)

  let rec toString eltToString s = StringUtils.listToString eltToString (toList s)

end
```

Figure 3: Skeleton of the `OperationTreeSet` module.

fleshing out the definitions of the `isEmpty`, `size`, `member`, `toList`, `fromList`, `toSexp`, and `fromSexp` operations.

Notes:

- You can test your implementation via the following OCAML commands:

  ```
  #cd "/students/username/cs251/ps4-group";;
  #use "load-optree-set.ml";;
  testZZZ();;
  ```

  where *ZZZ* is one of `Tiny`, `Small`, `Medium`, or `Large`. Before trying `testLarge()`, you should embiggen `;-)` the default OCAML stack size by following the instructions at the beginning of this problem.

  When a function fails a test, the nature of the problem may not always be apparent from the displayed feedback. Please study the testing code in `~/cs251/sets/SetTest.ml` or consult Lyn if you have trouble understanding the output of the tester.

- The testing code for all functions assumes that `fromList` works correctly, and the testing code for most functions (all except for `member`, and `toString`) assumes that `toList` works correctly. So you must implement `fromList` for any of the tests to work, and must implement `toList` for most of the tests to work.

- Your `toList` function should be defined by case analysis on the structure of the operation tree, as suggested by the skeleton for `toList` in Fig. 3. When fleshing out the `toList` definition, you will find it helpful to use functions in the `ListSetUtils` module. (These are also used in Handout #23 to implement `SortedListSet`.) The declaration

  ```
  module LSU = ListSetUtils
  ```

  in `OperationTreeSet` allows you to use the short prefix `LSU` rather than the long prefix `ListSetUtils` to access these functions.

- In `fromList`, for lists with ≥ 2 elements, you should first split the list into two (nearly) equal-length sublists and union the results of turning the sublists into sets. This yields a height-balanced operation tree.

- Your implementation of `member` should *not* use the `toList` function. Instead, it should be defined by case analysis on the structure of the operation tree, as suggested by the skeleton for `member` in Fig. 3.

- Your implementation of `size` and `isEmpty` *may* use the `toList` function. Indeed, it is difficult to implement these functions by a direct case analysis on the operation tree. Why?

- Before implementing `toSexp` and `fromSexp`, you should study the `toSexp` and `fromSexp` functions in the sorted list and BST implementations of sets.

- In `toSexp`, you should represent each non-empty node in the operation tree as an s-expression list whose first element is a lowercase symbol naming the operator and the rest of whose elements are the operands. An empty node should be represented as the symbol `empty` For example, the printed representation of the s-expression shown at the beginning of this problem is:

  ```
  (delete 4 (difference (union (union (insert 1 empty)
                                       (insert 4 empty))
                                (union (insert 7 empty)
                                       (insert 4 empty)))
                         (intersection (insert 1 empty)
                                       (union (insert 1 empty)
                                              (insert 6 empty)))))
  ```

Note that this printed representation is a legal OCAML expression that, when evaluated, would re-create the tree!

- In `fromSexp`, you can used nested patterns to succinctly describe how to convert s-expressions of the form described above into a constructor tree for the `set` datatype. If an inappropriate s-expression is encountered, `fromSexp` should raise an exception using the following code:

  ```
  raise (Failure ("OperationTreeSet.fromExp -- can't handle sexp:\n"
                  ^ (Sexp.sexpToString sexp)))
  ```

- The testing code tests `toSexp` and `fromSexp` together, so your implementation will not pass the test cases until both are working correctly.

Individual Problem Header Page
Please make this the first page of your hardcopy submission of individual problems.

CS251 Problem Set 4 Individual Problems
Due 6pm Friday, March 2

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.*

Part	**Time**	**Score**
General Reading		
Problem 1 [20]		
Total		

Group Problem Header Page
Please make this the first page of your hardcopy submission for group problems.

CS251 Problem Set 4 Group Problems
Due 6pm Friday, March 2

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.*

Part	**Time**	**Score**
General Reading		
Problem 1 [35]		
Problem 2 [30]		
Problem 3 [35]		
Total		