Problem Set 3
Due: 6pm Friday, February 25

Revisions: Feb 17: (1) In Group Problem 2, note that definitions and even some types of the functions are different from those of similarly named functions in Handout #20/ListUtils.ml; (2) In Group Problem 3a, the type of sum_multiples_of_3_or_5 should be int -> int -> int, not int * int -> int;

Overview:
The purpose of this assignment is to give you experience with first-class functions. These can twist your brain a bit, so leave sufficient time to do the problems.

Reading:
- Handout #16: The Substitution Model
- Handout #17: First-Class Functions
- Handout #20: Higher-Order List Functions
- Handout #21: Modules

Working Together:
Reminder: if you worked with a partner on PS1 or PS2 and want to work with a partner on this assignment, you must choose a different partner.

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 on the due date. The packet should include the individual problem header sheet and the final version of marbles.ml. Each student should also submit a softcopy (consisting of your final ps3-individual directory) to the drop directory ~cs251/drop/ps3/username, where username is your username. To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs251
cp -R ps3-individual ~cs251/drop/ps3/username/
```

Group Problem Submission:
Each team should turn in a single hardcopy submission packet for all group problems by slipping it under Lyn’s office door by 6pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the 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 to Problem 1;
3. your pencil-and-paper answers to Problem 2;
4. your final version of ps3-listfuns.ml for Problem 3;
5. your final version of PredSet.ml for Problem 4a;
6. your pencil-and-paper answers to Problem 4b;

Each team should also submit a single softcopy (consisting of your final ps3-group directory) to the drop directory ~cs251/drop/p3/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 [25]: Losing Your Marbles

This is an individual problem. Each student must solve this problem on her own without consulting any other person (except Lyn).

In this problem, you will define an OCAML function satisfying the following specification:

```ocaml
val marbles : int -> int -> int list list
```

Assume that \( m \) is a non-negative integer and that \( c \) is a positive integer. Given \( m \) marbles and a row of \( c \) cups, \( \text{marbles} \ m \ c \) returns a sorted list of all configurations whereby all \( m \) marbles are distributed among the \( c \) cups. Each configuration should be a list of length \( c \) whose elements are integers between 0 and \( m \) and the sum of whose elements is \( m \). The returned list should be ordered lexicographically (i.e., in “dictionary” order).

Some sample invocations of the \( \text{marbles} \) function are shown in Figs. 1 and 2.

Notes:

- Flesh out the skeleton of \( \text{marbles} \) in \(~\!/cs251/ps3-individual/marbles.ml\).
- Load your \( \text{marbles} \) function into OCAML via:
  ```
  #use "load-marbles.ml";;
  ```
  In addition to \( \text{marbles}.ml \), this will load some utility and testing files. Evaluate \( \text{testMarbles()} \) to test your \( \text{marbles} \) function on some sample inputs.
- As usual, you should use divide/conquer/glue as your problem-solving strategy. Strive to make your solution as simple as possible. For example, do not use more base cases than are strictly necessary.
- Feel free to define any auxiliary functions you find helpful.
- The \( \text{marbles} \) function may be defined without using any higher-order list functions. However, if you find such functions helpful and you are comfortable using them, you may use them in solving this problem. You may use any of the functions in the \text{ListUtils} or \text{FunUtils} modules in \(~\!/cs251/utils\). If you do use functions from these modules, you will need to use explicitly qualified names, such as \text{ListUtils.map} and \text{FunUtils.cons}.

```ocaml
# marbles 1 1;;
- : int list list = [[1]]
# marbles 1 2;;
- : int list list = [[0; 1]; [1; 0]]
# marbles 1 3;;
- : int list list = [[0; 0; 1]; [0; 1; 0]; [1; 0; 0]]
# marbles 2 1;;
- : int list list = [[2]]
# marbles 2 2;;
- : int list list = [[0; 2]; [1; 1]; [2; 0]]
# marbles 3 1;;
- : int list list = [[3]]
# marbles 3 2;;
- : int list list = [[0; 3]; [1; 2]; [2; 1]; [3; 0]]
```

Figure 1: Examples of the \( \text{marbles} \) function.
Figure 2: More examples of the `marbles` function.
Group Problems

Group Problem 1 [15]: Substitution Model

Consider the following OCAML declarations:

\[
\begin{align*}
\text{let sub x y = x - y} \\
\text{let app5 f = f 5} \\
\text{let flip f a b = f b a}
\end{align*}
\]

The substitution model introduced in lecture an in Handout #16 is able to show the step-by-step evaluation of OCAML expressions involving higher-order functions like app5 and flip. Here are two examples illustrating the substitution model (where we have delayed expanding the definitions of top-level variables until we need them):

\[
\begin{align*}
\text{# flip sub 3 2} \\
&\Rightarrow (\text{fun f a b -> f b a}) \text{ sub 3 2} \\
&\Rightarrow \text{ sub 2 3} \\
&\Rightarrow (\text{fun x y -> x - y}) \text{ 2 3} \\
&\Rightarrow 2 - 3 \\
&\Rightarrow -1
\end{align*}
\]

\[
\begin{align*}
\text{# flip flip 3 sub 5} \\
&\Rightarrow (\text{fun f a b -> f b a}) \text{ flip 3 sub 5} \\
&\Rightarrow \text{ flip sub 3 5} \\
&\Rightarrow (\text{fun f a b -> f b a}) \text{ sub 3 5} \\
&\Rightarrow \text{ sub 5 3} \\
&\Rightarrow (\text{fun x y -> x - y}) \text{ 5 3} \\
&\Rightarrow 5 - 3 \\
&\Rightarrow 2
\end{align*}
\]

Note that function application in OCAML is left-associative. For example, flip sub 3 2 is parsed as ((flip sub) 3) 2 and flip flip 3 sub 5 is parsed as (((flip flip) 3) sub) 2 The left associativity of function application dovetails nicely with the right associativity of the arrow type \(\rightarrow\) (think about this).

Use the substitution model to show the step-by-step evaluation of the following expressions. You may use OCAML to check your answers, but please do not do so until you have already tried to figure out the answers on your own.

a. [3] app5 (sub 1)

b. [4] app5 sub 2

c. [4] app5 (flip sub) 3

d. [4] flip app5 4 sub
Group Problem 2 [20]: Function Types

Figs. 3–4 contain twenty higher-order OCAML functions. For each function, write down the type that would be automatically reconstructed for it. For example, consider the following OCAML length function:

```ocaml
let rec length xs = 
  match xs with 
  | [] -> 0 
  | (_::xs') -> 1 + (length xs')
```

The type of this OCAML function is:

'\texttt{\texttt{\texttt{\texttt{\texttt{'a list -> int}}}}}

Notes:

- You can check your answers by typing them into the OCAML interpreter. But please write down the answers first before you check them — otherwise you will not learn anything!
- The particular type variable names used don’t matter, so any consistent renamings of the following types are also valid. For example, the type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b could also be written ('c -> 'e) -> ('q -> 'c) -> 'q -> 'e.
- Arrow types are right associative, so the type 'a -> 'b -> 'c -> 'd is parsed as if it were written 'a -> ('b -> ('c -> 'd))
- The product constructor * binds more tightly than ->, so 'a * 'b -> 'c * 'd means ('a * 'b) -> ('c * 'd) and not 'a * ('b -> 'c) * 'd
- A type variable (such as 'a or 'b) can be instantiated to any type in each use of a function with that type. For example, \texttt{n_fold} can be used with type \texttt{int -> (int -> int) -> int -> int} at one point of a program and \texttt{int -> (bool -> bool) -> bool -> bool} at another.
let id x = x

let compose f g x = (f (g x))

let rec repeated n f =
    if (n = 0) then id else compose f (repeated (n - 1) f)

let uncurry f (a,b) = (f a b)

let curry f a b = f(a,b)

let chPair x y = fun f -> f x y

let rec gen next isDone seed =
    if (isDone seed) then
        []
    else
        seed :: (gen next isDone (next seed))

let rec map f xs =
    match xs with
    | [] -> []
    | (x::xs') -> (f x) :: (map f xs')

let rec filter pred xs =
    match xs with
    | [] -> []
    | (x::xs') ->
        if (pred x) then
            x::(filter pred xs')
        else
            filter pred xs'

let product fs xs =
    map (fun f -> map (fun x -> (f x)) xs) fs

Figure 3: A sampler of higher-order functions in OCAML, part 1.
let rec zip pair =  
    match pair with  
    | ([]) , _ -> []  
    | (_, []) -> []  
    | (x::xs', y::ys') -> (x,y)::(zip(xs',ys'))

let rec unzip xys =  
    match xys with  
    | [] -> ([], [])  
    | ((x,y)::xys') ->  
        let (xs,ys) = unzip xys'  
        in (x::xs, y::ys)

let rec foldr binop init xs =  
    match xs with  
    | [] -> init  
    | (x::xs) -> binop x (foldr binop init xs)

let foldr2 ternop init xs ys =  
    foldr (fun (x,y) ans -> (ternop x y ans)) init (zip(xs,ys))

let flatten xss = foldr (@) [] xss

let rec forall pred xs =  
    match xs with  
    | [] -> true  
    | (x::xs') -> pred(x) && (forall pred xs')

let rec exists pred xs =  
    match xs with  
    | [] -> false  
    | (x::xs') -> (pred x) || (exists pred xs')

let rec some pred xs =  
    match xs with  
    | [] -> None  
    | (x::xs') -> if (pred x) then Some x else some pred xs'

let oneListOpToTwoListOp f =  
    let twoListOp binop xs ys = f binop (zip(xs,ys))  
    in twoListOp

let some2 pred = oneListOpToTwoListOp some pred

Figure 4: A sampler of higher-order functions in OCAMLO, part 2.
Group Problem 3 [45]: Higher-Order List Functions

Below you are asked to write several function declarations, most of which are curried versions of functions you implemented in Problem Set 2. The difference is that here you are not allowed to use explicit recursion to solve any of the problems. Instead, you should use the (mostly higher-order) list functions in utils/ListUtils.ml. A few other handy functions are defined in utils/FunUtils.ml. All definitions for this problem can be written in a few lines, and most can be written in one line.

Stubs for all the functions can be found in ps3-group/ps3-listfuns.ml. You should flesh out the definitions in this file. To try out your functions, execute the following in the OCAML interpreter:

```
#cd "/students/username/cs251/ps3-group"
#use "load-ps3-listfuns.ml"
```

This will load FunUtils.ml and ListUtils.ml in addition to your definitions from ps3-listfuns.ml.

You can test the functions in part x of this problem by evaluating the function invocation testx() in the OCAML interpreter (e.g., testa(), testb(), etc.). You can test all the functions on the assignment by evaluating the function invocation testall(). Note that even if your function passes all the test cases, it is not guaranteed to be correct; you are encouraged to extend the test cases in the testing file.

The file ps3-listfuns.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 List.map.

a. [5] val sum_multiples_of_3_or_5 : int -> int -> int

sum_multiples_of_3_or_5 m n returns the sum of all integers from m up to n (inclusive) that are multiples of 3 and/or 5. For example:

```
# sum_multiples_of_3_or_5 0 10;;
- : int = 33 (* 3 + 5 + 6 + 9 + 10 *)
# sum_multiples_of_3_or_5 (-9) 12;;
- : int = 22
# sum_multiples_of_3_or_5 18 18;;
- : int = 18
# sum_multiples_of_3_or_5 10 0;;
- : int = 0 (* The range "10 up to 0" is empty. *)
```

b. [5] val all_contain_multiple : int -> int list list -> bool

all_contain_multiple n nss returns true if each list of integers in nss contains at least one integer that is a multiple of n; otherwise it returns false.

```
# all_contain_multiple 5 [[17;10;12]; [25]; [3;7;5]];;
- : bool = true
# all_contain_multiple 3 [[17;10;12]; [25]; [3;7;5]];;
- : bool = false
# all_contain_multiple 3 [[]];
- : bool = true
```
c. \[5\] \textbf{val inner_product : int list -> int list -> int}

Assume that \(xs\) is the list of integers \([x_1, \ldots, x_n]\) and \(ys\) is the list of integers \([y_1, \ldots, y_n]\). (Both lists are assumed to have the same length \(n\).) Returns \(\sum_{i=1}^{n} x_i \cdot y_i\).

\[
\begin{align*}
\texttt{# inner_product [1;2;3] [4;5;6]} & \quad : \text{int} = 32 (* 4 + 10 + 18 *) \\
\texttt{# inner_product [] []} & \quad : \text{int} = 0
\end{align*}
\]

d. \[5\] \textbf{val alts : 'a list -> 'a list * 'a list}

Assume that the elements of a list are indexed starting with 1. \texttt{alts xs} returns a pair of lists, the first of which has all the odd-indexed elements (in the same relative order as in \(xs\)) and the second of which has all the even-indexed elements (in the same relative order as in \(xs\)).

\[
\begin{align*}
\texttt{# alts [7;5;4;6;9;2;8;3];} & \quad : \text{int list * int list} = ([7; 4; 9; 8], [5; 6; 2; 3]) \\
\texttt{# alts [7;5;4;6;9;2;8];} & \quad : \text{int list * int list} = ([7; 4; 9; 8], [5; 6; 2]) \\
\texttt{# alts [7];} & \quad : \text{int list * int list} = ([7], []) \\
\texttt{# alts [];} & \quad : '_a list * '_a list = ([], [])
\end{align*}
\]

e. \[5\] \textbf{val cartesian_product : 'a list -> 'b list -> ('a * 'b) list}

cartesian\_product \(xs\ ys\) returns a list of all pairs \((x, y)\) where \(x\) ranges over the elements of \(xs\) and \(y\) ranges over the elements of \(ys\). The pairs should be sorted first by the \(x\) entry (relative to the order in \(xs\)) and then by the \(y\) entry (relative to the order in \(ys\)).

\[
\begin{align*}
\texttt{# cartesian_product [1; 2] ['a'; 'b'; 'c'];} & \quad : (\text{int * char}) list = [(1, 'a'); (1, 'b'); (1, 'c'); (2, 'a'); (2, 'b'); (2, 'c')] \\
\texttt{# cartesian_product [2; 1] ['a'; 'b'; 'c'];} & \quad : (\text{int * char}) list = [(2, 'a'); (2, 'b'); (2, 'c'); (1, 'a'); (1, 'b'); (1, 'c')] \\
\texttt{# cartesian_product ['c'; 'a'; 'b'] [2; 1];} & \quad : (\text{char * int}) list = [(‘c’, 2); (‘c’, 1); (‘a’, 2); (‘a’, 1); (‘b’, 2); (‘b’, 1)] \\
\texttt{# cartesian_product [1] ['a'];} & \quad : (\text{int * char}) list = [(1, 'a')] \\
\texttt{# cartesian_product [] ['a'; 'b'; 'c'];} & \quad : ('_a * char) list = []
\end{align*}
\]
f. \[5\] \textbf{val} \ bits : \textit{int} \rightarrow \textit{int} \textit{list}

\textit{bits} \ \textit{n} \ \text{returns a list of the bits (0s and 1s) in the binary representation of} \ \textit{n}.

\begin{verbatim}
# bits 5;;
- : int list = [1; 0; 1]
# bits 10;;
- : int list = [1; 0; 1; 0]
# bits 11;;
- : int list = [1; 0; 1; 1]
# bits 22;;
- : int list = [1; 0; 1; 1; 0]
# bits 23;;
- : int list = [1; 0; 1; 1; 1]
# bits 46;;
- : int list = [1; 0; 1; 1; 1; 0]
# bits 0;;
- : int list = [0] (* special case! *)
\end{verbatim}

\hfill 

g. \[5\] \textbf{val} \ n\textit{_fold} : \textit{int} \rightarrow ('a \rightarrow 'a) \rightarrow 'a \rightarrow 'a

\textit{n\textit{\textunderscore}fold} \ \textit{n} \ \textit{f} \ \text{returns the} \ \textit{n}-fold \ \text{composition of the function} \ \textit{f}.

\begin{verbatim}
# n_fold 5 ((+) 1) 0;;
- : int = 5
# n_fold 5 (( *) 2) 1;;
- : int = 32
# n_fold 3 ((flip (/)) 2) 100;;
- : int = 12
# n_fold 0 ((flip (/)) 2) 100;;
- : int = 100
\end{verbatim}

\hfill 

h. \[5\] \textbf{val} \ inserts : 'a \rightarrow 'a \textit{list} \rightarrow 'a \textit{list} \textit{list}

\textit{Assume that ys} \ \text{is a list with} \ \textit{n} \ \text{elements.} \ \textit{insert} \ (x,ys) \ \text{returns a} \ \textit{n} + 1-length \ \text{list of lists} \ \textit{showing all ways to insert a single copy of} \ \textit{x} \ \textit{into} \ \textit{xs}.

\begin{verbatim}
# inserts 3 [5;7;1];;
- : int list list = [[3; 5; 7; 1]; [5; 3; 7; 1]; [5; 7; 3; 1]; [5; 7; 1; 3]]
# inserts 3 [5;3;1];;
- : int list list = [[3; 5; 3; 1]; [5; 3; 3; 1]; [5; 3; 3; 1]; [5; 3; 1; 3]]
# inserts 3 [];;
- : int list list = [[3]]
\end{verbatim}

\textit{Hint: It is possible to solve this problem with foldr, but it is easier to use foldr’}.
i. \[5\] val permutations : 'a list -> 'a list list

Assume that \(xs\) is a list of distinct elements (i.e., no duplicates). \texttt{permutations} \(xs\) returns a list of all the permutations of the elements of \(xs\). The order of the permutations does not matter.

```ocaml
# permutations [];;
- : '_a list list = [[]]
# permutations [4];;
- : int list list = [[4]]
# permutations [3;4];;
- : int list list = [[3; 4], [4; 3]]
# permutations [2;3;4];;
- : int list list = [[2; 3; 4], [3; 2; 4], [2; 4; 3], [4; 2; 3], [4; 3; 2]]
```

Group Problem 4 [20]: 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 \texttt{SET} signature (Fig. 5) used in Handout #21, we will use the somewhat different \texttt{PRED_SET} signature shown in Fig. 6. Here is a comparison of \texttt{PRED_SET} with \texttt{SET}:

- It has the same \texttt{empty}, \texttt{singleton}, \texttt{member}, \texttt{union}, \texttt{intersection}, \texttt{difference}, and \texttt{fromList} operations as \texttt{SET}.
- It does not support the \texttt{toList} or \texttt{toString} operations of \texttt{SET}.
- It has two operations that \texttt{SET} does not have: \texttt{fromPred} and \texttt{toPred}. These allow converting between predicates and sets.

The \texttt{fromPred} and \texttt{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:

```ocaml
# let ps1 = fromPred (fun x -> (x = 2) || (x = 3) || (x = 5));;
val ps1 : int PredSet.set = <abstr>
# member 3 s;;
- : bool = true
# member 5 s;;
- : bool = true
# member 4 s;;
- : bool = false
```

11
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 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 (* intersection 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 toString : ('a -> string) -> 'a set -> string (* string representation of the set *)
end

Figure 5: 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 (* create a set from a list *)
  val toList: 'a set -> 'a list (* list all set elts, sorted low to high *)
  val toPred: ('a -> string) -> 'a set (* string representation of the set *)
end

Figure 6: A signature for a version of sets based upon predicates.
The set \( \text{ps1} \) consists of exactly those elements satisfying the predicate passed to \( \text{fromPred} \) – 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:

\[
\text{fromPred} \left( \text{fun } x \to (x \mod 2) = 0 \right)
\]

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 \( \text{fromPred} \left( \text{fun } x \to \text{true} \right) \). 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:

\[
\text{fromPred} \left( \text{fun } x \to (x \geq 251) \&\& (x \leq 6821) \right)
\]

**a. [12]: PredSet**

The most obvious way to implement the \texttt{PRED\_SET} signature is in a module \texttt{PredSet} that defines the \texttt{set} type as a predicate:

\[
\texttt{type 'a set = 'a -> bool}
\]

Based on this representation, flesh out all the function definitions in the the \texttt{PredSet} module in the file `~/cs251/ps3-group/PredSet.ml`. Each of your definitions should be a one-liner. (For \texttt{fromList}, you may use operations from the \texttt{List} module.) Use \#use "PredSet.ml" to load your module and \texttt{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.

**b. [8]: Other Functions**

In this problem, you are asked to consider whether it is possible to implement the \texttt{SET} and \texttt{PRED\_SET} signatures if we extend them with additional functionality. Explain all your answers.

1. Can we add to the \texttt{PRED\_SET} signature the following function?

   \[
   \texttt{val toList: 'a set -> 'a list}
   \]

   Returns a list of all the elements in set.

2. Can we add to the \texttt{SET} signature the following function?

   \[
   \texttt{val fromPred: ('a -> bool) -> 'a set}
   \]

   Returns a set of all elements satisfying the given predicate.

3. Can we add the following function to the \texttt{SET} signature? To the \texttt{PRED\_SET} signature?

   \[
   \texttt{val isEmpty: 'a set -> bool}
   \]

   Returns \texttt{true} if the set is empty, and \texttt{false} otherwise.

4. Can we add the following function to the \texttt{SET} signature? To the \texttt{PRED\_SET} signature?

   \[
   \texttt{val complement: 'a set -> 'a set}
   \]

   Returns the complement of the given set – i.e., all the value of type \texttt{'a} that are not in the given set.

5. Can we add the following function to the \texttt{SET} signature? To the \texttt{PRED\_SET} signature?

   \[
   \texttt{val isSubset: 'a set -> 'a set -> bool}
   \]

   Returns \texttt{true} if all of the elements of the first set parameter are are also elements of the second set parameter, and \texttt{false} otherwise.

13
Extra Credit 1 [15]: Church Predicates

The First-Class Functions handout (#17) discusses how \( n \)-fold composition functions (so-called Church numerals) can be viewed as the basis of a system for arithmetic. Write an OCAML definition for the `pred` function that is described in the handout. Add this definition to the file `~/cs251/ps3-group/church.ml`, which includes other definitions from Handout #17. Evaluate `#use "load-church.ml"` to load the file into ocaml.

- Your definition should not use any of the following: `n_fold`, `int2ch`, `ch2int`, or any recursively defined function.

- Your definition may use any other functions. In particular, the following functions may be useful: `nonce`, `succ`, `chPair`, `chFst`, and `chSnd`.

- Although solutions can be very short, this is a very challenging problem. *Hint*: One approach to defining `pred` is to use an iteration on pairs.
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(s):

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 1 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [45]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4 [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>