Problem Set 4  
Due: 6pm Wednesday, February 27

This is the final version of PS4 with all problems.

Overview:
The individual problem on this assignment tests your understanding of higher-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:
- Chapters 6, 12 (ignore 12.4), and 13 of Hickey’s Introduction to the Objective Caml Programming Language
- Handout #17: Modules and Data Abstraction in OCAML.
- Handout #18: John Backus’s Turing Award Lecture (Sections 1 – 11 and 15–16)
- Handout #19: Dean & Ghemawat’s MapReduce paper
- Handout #21: 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 on the due date. The packet should include:
1. an individual problem header sheet;
2. your final version of partition.ml from Problem 1;
3. a transcript showing the results of running test_partition() and test_isPartitioning().

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 6pm on the due date. 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 [30]: Partitioning

Background

Recall that an equivalence relation \( R \) on a set \( T \) is a binary relation on \( T \) (i.e., a subset of \( T \times T \)) that is reflexive, symmetric, and transitive. Also recall that an equivalence relation \( R \) on \( T \) partitions any subset \( S \subseteq T \) into equivalence classes \( S_1, \ldots, S_n \) such that:

1. Each \( S_i \) is nonempty;
2. \( S = \bigcup_{i=1}^{n} S_i \); and
3. if \( x \in S_i, y \in S_j \), and \( xRy \) (i.e., \( x \) and \( y \) are related by \( R \)), then \( i = j \). This condition says that only elements in the same equivalence class can be related by \( R \).

Each equivalence class is also known as a partition relative to \( R \), and the collection of equivalence classes is called a partitioning of \( S \).

In this problem you will write functions that involve partitioning OCAML lists into list of lists relative to equivalence relations phrased as curried binary functions written in OCAML. Here are some examples of such equivalence relations:

- \((=)\): The (polymorphic) OCAML equality operator is an equivalence relation for any type, e.g., \texttt{int, char, string, int * char, int list, ...}

- \texttt{eqMod n}, where \texttt{eqMod} is defined as

  ```ocaml
  let eqMod n x y = (((max x y) - (min x y)) mod n) = 0
  ```

determines if two integers are equal modulo an integer \( n \geq 1 \).

- \texttt{pairSumEq}, which is defined as

  ```ocaml
  let pairSumEq (a,b) (c,d) = (a+b) = (c+d)
  ```

determines if two pairs of integers have components that sum to the same value.

- \texttt{sameLen}, which is defined as

  ```ocaml
  let sameLen s1 s2 = (String.length s1) = (String.length s2)
  ```

determines if two strings have the same length

- \texttt{sameFirstChar}, which is defined as

  ```ocaml
  let sameFirstChar s1 s2 =
  if (s1 = "") || (s2 = ") then
    s1 = s2
  else
    (String.get s1 0) = (String.get s2 0)
  ```

determines if two strings have the same first character (or if both strings are empty).

Your Task

In this problem, you will define two functions. You may define them in either order.
a. In this part, you will define the following \texttt{partition} function:

\begin{verbatim}
val partition: ('a -> 'a -> bool) -> 'a list -> 'a list list

Suppose \texttt{eq} is an equivalence relation expressed as a curried binary OCAML predicate on elements of type \texttt{'a} and \texttt{x}s is a list of elements of type \texttt{'a}. Then \texttt{partition eq xs} returns a list of the equivalence classes of \texttt{xs} relative to \texttt{eq}, where each equivalence class is represented as a list of elements from \texttt{xs}. Within each equivalence class, elements must appear in the same relative order as within \texttt{xs}. However, the equivalence classes themselves may appear in any order.

For example:

\begin{verbatim}
# partition (eqMod 3) (range (-5) 5);
- : int list list = [[-3; 0; 3]; [-5; -2; 1; 4]; [-4; -1; 2; 5]]
\end{verbatim}

In this example, \([-3; 0; 3]\) is the equivalence class of integers between -5 and 5 that are 0 mod 3; \([-5; -2; 1; 4]\) are the numbers that are 1 mod 3; and \([-4; -1; 2; 5]\) are the numbers that are 2 mod 3.\(^1\) Because the input list contains integers ordered low to high, each sublist of the result must also be ordered low to high. However, the order of the sublists is immaterial.

The five other permutations of the above result would also be correct results:

\begin{verbatim}
[[-3; 0; 3]; [-4; -1; 2; 5]; [-5; -2; 1; 4]]
[[-4; -1; 2; 5]; [-5; -2; 1; 4]; [-3; 0; 3]]
[[-5; -2; 1; 4]; [-3; 0; 3]; [-4; -1; 2; 5]]
[[-5; -2; 1; 4]; [-4; -1; 2; 5]; [-3; 0; 3]]
\end{verbatim}

Here are some more examples of \texttt{partition}:

\begin{verbatim}
# partition (eqMod 2) (range (-5) 5);
- : int list list = [[-4; -2; 0; 2; 4]; [-5; -3; -1; 1; 3; 5]]
\end{verbatim}

\begin{verbatim}
# partition (eqMod 4) (range (-5) 5);
- : int list list = [[-2; 2]; [-5; -1; 3]; [-4; 0; 4]; [-3; 1; 5]]
\end{verbatim}

\begin{verbatim}
# partition (eqMod 5) (range (-5) 5);
- : int list list = [[-4; 1]; [-3; 2]; [-2; 3]; [-1; 4]; [-5; 0; 5]]
\end{verbatim}

\begin{verbatim}
# partition (eqMod 8) (range (-5) 5);
- : int list list = [[-2]; [-1]; [0]; [1]; [2]; [-5; 3]; [-4; 4]; [-3; 5]]
\end{verbatim}

\begin{verbatim}
# partition pairSumEq (flatten (map (fun x -> (map (pair x) (range 0 3))) (range 0 3))));
- : (int * int) list list =
  [[(0, 0)]; [(0, 1)]; [(1, 0)]; [(0, 2)]; [(1, 1)]; [(2, 0)]; [(0, 3)]; [(1, 2)]; [(2, 1)]; [(3, 0)]; [(1, 3)]; [(2, 2)]; [(3, 1)]; [(2, 3)]; [(3, 2)]; [(3, 3)]
\end{verbatim}

\begin{verbatim}
# partition sameLen ["she"; "said"; "that"; "he"; "saw"; "her"; "in"; "the"; "store"];
- : string list list =
  [["said"; "that"]; ["he"; "in"]; ["she"; "saw"; "her"; "the"]; ["store"]]
\end{verbatim}

\begin{verbatim}
# partition sameFirstChar ["she"; "said"; "that"; "he"; "saw"; "her"; "in"; "the"; "store"];
- : string list list =
  [["he"; "her"]; ["in"]; ["that"; "the"]; ["she"; "said"; "saw"; "store"]]
\end{verbatim}
\end{verbatim}

\(^1\)The notion of “mod” used here is the traditional mathematical notion of mod, in which \(x \ mod \ n\) denotes a number between 0 and \(n - 1\) as long as \(n \geq 1\). This is different than OCaml’s \texttt{mod} function, which in this situation will return a negative result if \(x\) is negative. This is the reason why \texttt{eqMod n x y} subtracts the min of \(x\) and \(y\) from the max of \(x\) and \(y\): the result of this subtraction is always nonnegative.
# partition (=) [3;2;-1;3;1;3;1;2;1;-1;0;3];
- : int list list = [[2; 2]; [1; 1; 1]; [-1; -1]; [0]; [3; 3; 3; 3]]

The final example shows that partition must preserve any duplicate elements in the input list in the resulting partitions.

Your task in this part is to flesh out a definition for the partition function, which can be found in the file ~/cs251/ps4-individual/partition.ml. Your definition should not use any explicit 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-partition.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 partition.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 partition.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.
- Partial credit will be awarded if the equivalence classes contain the correct elements, but not in the correct order.
- Your definition need not be particularly efficient; for example, it might examine xs several times or might perform certain comparisons multiple times. However, your definition should not be unreasonably inefficient. In particular, it should not have a worst-case running time more than quadratic in the length of xs.
- Depending on how you define partition, you might see warnings like the following for nonexhaustive pattern-matching:
  File "partition.ml", line 25, characters 20-47:
  Warning P: this pattern-matching is not exhaustive.
  Here is an example of a value that is not matched:
  []
This warning will arise when you use pattern-matching on a list you know to be nonempty without providing a clause for the empty case. Such warnings are OK and you can safely ignore them.
- Use test_partition() to test your partition function on a sample testing suite. After any change to your definition of partition, you should reload all files via
  #use "load-partition.ml"
before invoking test_partition(). (Otherwise you will be testing the old definition, not the new one!)
b. [18] In this part, you will define the following isPartitioning function:

```ocaml
val isPartitioning : ('a -> 'a -> bool) -> 'a list list -> bool
```

Suppose `eq` is an equivalence relation expressed as a curried binary OCAML predicate on elements of type `'a` and `xss` is a list of lists of elements of type `'a`. Then `isPartitioning eq xss` returns `true` if `xss` is a partitioning of some list relative to `eq` and otherwise returns `false`. That is, `isPartitioning eq xss` returns `true` iff (1) every sublist of `xss` is nonempty; (2) in every sublist of `xss`, all elements are related by `eq`; and (3) no two elements from different sublists of `xss` are related by `eq`.

For example:

```ocaml
# isPartitioning (eqMod 3) [[0; 3; 6; 9]; [1; 4; 7]; [2; 5; 8]];;
- : bool = true

# isPartitioning (eqMod 3) [[3; 9; 6; 0]; [4; 1; 7; 1]; [2; 2; 5; 8]];;
- : bool = true

# isPartitioning (eqMod 3) [[0; 3; 6; 9]; [1; 4; 7]; [ ]; [2; 5; 8]];;
- : bool = false (* One of the sublists is empty *)

# isPartitioning (eqMod 3) [[0; 3; 6; 9]; [1; 4; 7]; [2; 5; 8; 10]];;
- : bool = false (* 10 is unrelated to the other elements of the final sublist *)

# isPartitioning (eqMod 3) [[0; 3]; [1; 4; 7]; [6; 9]; [2; 5; 8]];;
- : bool = false (* 0, 3, 6, and 9 should be in one partition, not two *)
```

Your task in this part is to flesh out a definition for the `isPartitioning` function, which can be found in the file `~/cs251/ps4-individual/partition.ml`. Your definition should not use any explicit recursion but should instead use the higher-order list functions in the `ListUtils` module.

**Notes:**

- The general notes from Part a (those not specific to the `partition` function itself) apply here as well.
- Your `isPartitioning` function should take time no longer than quadratic in the total number of elements.
- Use the properties of equivalence relations to avoid performing unnecessary comparisons. For example, to show that all elements of a list `xs` are related by an equivalence relation `eq`, you do not need to compare all pairs of elements in the list. Instead, just compare all elements to the first element of the list. If all elements are related by `eq` to the first element, then by symmetry and transitivity, all the elements are related to each other. Similarly, for showing that no two elements from different sublists of `xss` are related by `eq`, you need only perform comparisons involving the `first` element from each sublist of `xss`.
- Use `test_isPartitioning()` to test your `isPartitioning` function on a sample testing suite.
Group Problems

Group Problem 1 [45]: 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:
  
  ```ocaml
  if p_1 then e_1
  else ...
  else if p_n then p_n
  else e_{n+1}
  ```

- $(e_1, \ldots, e_n)$ denotes the sequence of the $n$ values of the expressions $e_1, \ldots, e_n$. $\emptyset$ denotes the empty sequence. Because FP is dynamically typed, such sequences can represent both tuples and lists from OCAML.

- The symbol $\bot$ (pronounced “bottom”) denotes the value of an expression that doesn’t terminate (i.e., it loops infinitely) or terminates with an error.

- If $f$ is a function and $x$ is an object (atom or sequence of objects), then $f : x$ denotes the result of applying $f$ to $x$.

- $[f_1, \ldots, f_n]$ is a functional form denoting a sequence of $n$ functions, $f_1$ through $f_n$. The application rule for this functional form is $[f_1, \ldots, f_n] : x = (f_1 : x, \ldots, f_n : x)$ — i.e., the result of applying a sequence of $n$ functions to an object $x$ is an $n$-element sequence consisting of the results of applying each of the functions in the function sequence to $x$.

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

a. [25] Answer the following questions about Backus’s paper. Your answers should be concise but informative.

i. 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.

ii. Many programming languages have at least two syntactic categories: expressions and statements. Backus claims that expressions are good but statements are bad. Explain his claim.

iii. In Sections 6, 7, and 9 of the paper, Backus discusses three problems/defects with von Neumann languages. Summarize them.

iv. What are applicative languages and how do they address the three problems/defects mentioned by Backus for von Neumann languages?

v. 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?

vi. Backus wrote this paper long before the development of JAVA and OCAML. Based on his paper, how do you think he would evaluate these two languages?
b. [12] Consider the following FP definition:

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

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

c. [8] The Dean & Ghemawat paper describes the MapReduce framework for manipulating large datasets. Evaluate this framework. In what ways is it like an applicative language? In what ways is it still like a von Neumann language?

**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 \texttt{SET} signature used in Handout #17 (see Fig. 1)\(^2\) we will use the somewhat different \texttt{PRED_SET} signature shown in Fig. 2. Here is a comparison of \texttt{PRED_SET} with \texttt{SET}:

- **\texttt{PRED_SET}** has many of the same operations as \texttt{SET}: \texttt{empty}, \texttt{singleton}, \texttt{member}, \texttt{union}, \texttt{intersection}, \texttt{difference}, and \texttt{fromList}

- **\texttt{PRED_SET}** does not support the following operations of \texttt{SET}: \texttt{isEmpty}, \texttt{size}, \texttt{toList}, \texttt{fromSexp}, \texttt{toSexp}, or \texttt{toString}.

- **\texttt{PRED_SET}** has two operations that \texttt{SET} does not have: \texttt{fromPred} and \texttt{toPred}. These allow converting between predicates and sets.

\footnotesize

```ocaml
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 (* 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 fromSexp : (Sexp.sexp -> 'a) -> 'a set (* translates s-expression rep. into set *)
  val toSexp : ('a -> Sexp.sexp) -> 'a set (* translates set into s-expression rep. *)
  val toString : ('a -> string) -> 'a set (* string representation of the set *)
end
```

Figure 1: The \texttt{SET} signature.

\footnotesize\(^2\)The \texttt{SET} signature in Fig. 1 includes two functions, \texttt{fromSexp} and \texttt{toSexp} that were not mentioned in Handout #17, but are actually part of the set implementation you will use in Group Problem 3.
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:

```ocaml
# 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:

```ocaml
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:

```ocaml
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:

```ocaml
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

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

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

```ocaml
fun y -> expression determining if y is in the set
```
For example, you can write the union definition as

```ml
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. Note: additional functionality is impossible to implement iff it is uncomputable a la CS235.

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

   ```ml
   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?

   ```ml
   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 SET signature? To the PRED_SET signature?

   ```ml
   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?

   ```ml
   val complement: 'a set -> 'a set
   Returns the complement of the given set – i.e., all the values 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?

   ```ml
   val isSubset: 'a set -> 'a set -> bool
   Returns true if all of the elements of the first set parameter are also elements of the second set parameter, and false otherwise.
   ```
Group Problem 3 [25]: 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 #17 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 \( s \), \( \text{fromSexp (toSexp } s \) \) 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 (#17). The code for this implementation is in `~/cs251/sets/SortedListSet.ml`. You can test it via the following OCAML commands:

```ocaml
#cd "~/cs251/sets/SortedListSet.ml";
#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:

```ocaml
#cd "~/cs251/sets/BSTSet.ml";
#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:

   ```bash
   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:

```ocaml
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:

![Operation Tree Representation](image)

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 trivial and have already been implemented. You are responsible for 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:
  ```ocaml
  #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.
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 []
    | Intersection(s1, s2) -> Sexp.Seq []
    | Difference(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.
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 #17 to implement `SortedListSet`.) The declaration

```ocaml
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 \( \geq 2 \) 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 `toList` and `fromList`, you should study the `toList` and `fromList` functions in the sorted list and BST implementations of sets.

In `toList`, 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 `fromList`, 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, `fromList` should raise an exception using the following code:

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

The testing code tests `toList` and `fromList` together, so your implementation will not pass the test cases until both are working correctly.
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 1 [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CS251 Problem Set 4 Group Problems
Due 6pm Wednesday, February 27

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 [45]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [30]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>