**CS251 Programming Languages** **Handout # 17**
**Prof. Lyn Turbak** **February 16, 2004**
**Wellesley College**

# Problem Set 3
## Due: 6pm Saturday, February 21

**Overview:**

The purpose of this assignment is to give you practice with sum-of-product datatypes, s-expressions, and modules in OCaml. You will do this in the context of implementing a set datatype in three different ways.

**Reading:**

- Handout #8 (Jason Hickey's OCaml tutorial): Chapters 6, 7, 10 (ignore 10.4), and 11.
- Handout #15: Datatypes and Data Abstraction in OCaml.

**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.

**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 (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 final version of `BSTSet.ml` for Problem 1;
3. your final version of `OperationTreeSet.ml` for Problem 2;
4. your final version of `PredSet.ml` for Problem 3a;
5. your pencil-and-paper answers to Problem 3b;

Each team should also submit a single softcopy (consisting of your final `ps3` 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). 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 ~cs251/drop/ps3/username/
```

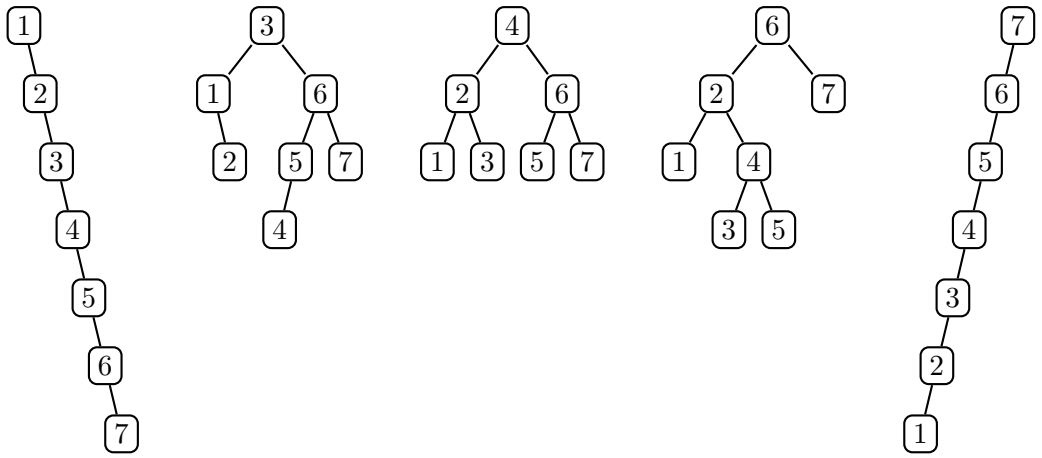
**Problem 1 [50]:** `BSTSet`

In this problem, you will flesh out an implementation of sets using a binary search tree (BST) representation. Your implementation should match the `SET` signature presented in Fig. 1, which is discussed in Handout #15. To represent binary trees, you should use the `Bintree` module discussed in Handout #15, whose signature is presented in fig:bintree-sig.

Recall that a BST is a binary tree in which the following BST conditions are satisfied at every non-leaf node `Node(l,v,r)`:

- all values in `l` are strictly less than `v`.
- all values in `r` are strictly greater than `v`.

We assume that all orderings are determined via OCAML's implicit value ordering using `<`, `>`, etc. For example, below are some of the many possible BSTs containing the numbers 1 through 7. Note that a BST is not required to be balanced in any way.




You should write all your definitions in the skeleton of the `BSTSet` module, which is in the file `~/cs251/ps3/BSTSet.ml`. This module has the form:

```
module BSTSet : SET = struct
  open Bintree
  type 'a set = 'a bintree
  function declarations
end
```

The declaration `open Bintree` makes all the declarations in the `Bintree` module (including the `bintree` datatype, the `Leaf` and `Node` constructors, and all the functions in the `BINTREE` signature) available without qualification. That is, you can write `Node` rather than `Bintree.Node`, `map` rather than `Bintree.map`, etc.

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

Figure 1: The SET signature.

```
module type BINTREE = sig
  type 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree
  val int_tree : int bintree
  val string_tree : string bintree
  val map : ('a -> 'b) -> 'a bintree -> 'b bintree
  val fold : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b bintree -> 'a
  val nodes : 'a bintree -> int
  val height : 'a bintree -> int
  val sum : int bintree -> int
  val prelist : 'a bintree -> 'a list
  val inlist : 'a bintree -> 'a list
  val postlist : 'a bintree -> 'a list
  val toString : ('a -> string) -> 'a bintree -> string
end
```

Figure 2: The BINTREE signature.

*Notes:*

- As always, begin the problem set with the following Linux commands:

```
cd ~/cs251
cvs update -d
```

- Before beginning your coding, you might wish to review how elements are inserted and deleted from BSTs by consulting your CS230 notes or any of the Java data structure books in SCI 173. In particular, deletion of the value of a node with two non-leaf subtrees is tricky and requires careful handling.

- In your function declarations, you may refer to other functions in `BSTSet` as long as they are either (1) defined above the current function or (2) defined in the same recursive collection as the current function. (This is particularly helpful in `union`, `intersection`, and `difference`, where it is helpful to use the `toList` function to process all of the elements in one of the set parameters.) You may need to reorder the declarations or organize them into recursive collections to make this happen.

- You may define any auxiliary functions you find helpful. Make sure that these are defined above where you need them or in the same recursive scope where you need them. Because of signature matching in the declaration `module BSTSet : SET =` ..., you will not be able to refer to any of your auxiliary functions outside the module. If you want to test your auxiliary functions, you will need to temporarily remove the `: SET` from the declaration. (But remember to put it back later!).

- Because signature matching makes the `set` type abstract, you cannot inspect the tree structure of your examples unless you temporarily remove the `: SET` from the declaration of `BSTSet`.

- You are welcome to use functions from other modules, especially the `List` module. You will need to explicitly qualify such references – e.g., `List.map`.

- The `toString` function should use the `toString` function of the `Bintree` module. For example, suppose that `s` is the set defined as follows:

```
let set = insert 3 (insert 7 (insert 9 (insert 1 (singleton 5))))
```

  Then `BSTSet.toString string_of_int s` should return the string:

```
"((* 1 (* 3 *)) 5 ((* 7 *) 9 *))"
```

- In `toSexp` and `fromSexp`, the following s-expression notation for BSTs should be used:

  - A (top-level) leaf is represented as `()`.
  - The default representation of a node is `(`*`lsexp vsexp rsexp`*`)`, where *vsexp* is the s-expression notation for the node value and *lsexp* and *rsexp* are s-expression notations for the left and right subtrees. The following optimizations are used to reduce the size of node representations:
    - If both the left and right subtrees of a node are leaves, then the node is represented as just *vsexp*.[1]

[1] It is assumed that *vsexp* is not delimited by parens, else the notation might be ambiguous.

* If exactly one of the left or right subtrees of a node is a leaf, then the representation for the leaf subtree is omitted from (*lsexp vsexp rsexp*) — i.e., it should have the form (*vsexp rsexp*) or (*lsexp vsexp*).

For example, for the example `s` mentioned above, we have:

```
# StringUtils.print (Sexp.sexpToString (toSexp (fun x -> Sexp.Int x) s));;
((1 (3)) 5 ((7) 9))- : unit = ()
```

We can test `fromSexp` as follows:

```
# toList (fromSexp (fun sexp -> match sexp with
                                  Sexp.Int i -> i
                                | _ -> raise (Failure "wrong form"))
          (Sexp.stringToSexp  "((1 (3)) 5 ((7) 9))"));;
- : int list = [1; 3; 5; 7; 9]
```

- To load the `BSTSet` module and the modules it depends on, execute the following OCAML commands:

  ```
  #cd "/home/your-username/cs251/ps3";;
  #use "load-sets.ml";;
  ```

  Executing the second line will cause lots of declarations to be displayed on the screen. **Verify that the declaration `module BSTSet : SET` is included in the declarations** (near the end, right after `module SortedListSet : SET`). If this declaration does not appear and is instead replaced by an error message, it means that your `BSTSet` module has a syntax error or type error. You must fix any such errors before you attempt to test your module.

- When testing your `BSTSet` functions by hand in the top-level interpreter, by default you must use fully qualified references to the `BSTSet` functions, as in the following example:

  ```
  # let s = BSTSet.insert 3 (BSTSet.insert 7 (BSTSet.insert 9
              (BSTSet.insert 1 (BSTSet.singleton 5))));;
  val s : int BSTSet.set = <abstr>
  ```

  You can shorten this considerably if you first open the `BSTSet` module within the top-level interpreter:

  ```
  # open BSTSet;;
  # let s = insert 3 (insert 7 (insert 9 (insert 1 (singleton 5))));;
  val s : int BSTSet.set = <abstr>
  ```

  **Warning:** To use the abbreviations, you *must* reopen the `BSTSet` module after every time you execute `#use "load-sets.ml"`. Furthermore, you *must* redefine any definitions (like `s`) every time you reload `load-sets.ml` (which is why its more convenient to put any such definitions in a file). Failure to observe these points can lead to confusing interactions in which your expressions are refering to out-of-date definitions rather than the current ones. For example, study the interactions in Fig. 3.

- You can perform a suite of automatic tests on your `BSTSet` implementation by executing `BSTSetTest.testSmall()` or `BSTSetTest.testBig()`. These read in a set of words from some files and peform various set manipulations that are compared to the results of a standard reference implementation. For example, see Fig. 4.

  The code for these tests is in `~/cs251/sets/SetTest.ml`. You do not need to study this code in order to use it (but may want to in order to better understand how it works).

```
#use "load-sets.ml";;
(* lots of declarations omitted. *)
# open BSTSet;;
# let s = insert 3 (insert 7 (insert 9 (insert 1 (singleton 5))));;
val s : int BSTSet.set = <abstr>
# StringUtils.print (Sexp.sexpToString (toSexp (fun x -> Sexp.Int x) s));;
((9 (7)) 5 ((3) 1))- : unit = () (* Oops, left and right subtrees are swapped *)
(* Edit the source code to fix the bug, and then reload *)
#use "load-sets.ml";;
(* lots of declarations omitted. *)
# StringUtils.print (Sexp.sexpToString (toSexp (fun x -> Sexp.Int x) s));;
((9 (7)) 5 ((3) 1))- : unit = ()
(* Bug is still there! Ah -- forget to reopen BSTSet and redefine s *)
# open BSTSet;;
# let s = insert 3 (insert 7 (insert 9 (insert 1 (singleton 5))));;
val s : int BSTSet.set = <abstr>
# StringUtils.print (Sexp.sexpToString (toSexp (fun x -> Sexp.Int x) s));;
((1 (3) 5 ((7) 9)))- : unit = () (* Now test shows that bug is fixed *)
```

Figure 3: Interactions highlighting the importance of reopening modules and redefining examples after reloading modules.

```
# BSTSetTest.testSmall();;
Reading ../text/green-eggs-init.txt into list ...done
List has 43 elements
Creating set from list ...done
Reading ../text/cat-in-hat-init.txt into list ...done
List has 66 elements
Creating set from list ...done
Reading ../text/green-eggs-init.txt into list ...done
List has 43 elements
Creating set from list ...done
Reading ../text/cat-in-hat-init.txt into list ...done
List has 66 elements
Creating set from list ...done
Testing insert ...OK!
Testing delete ...OK!
Testing union ...OK!
Testing intersection ...OK!
Testing difference ...OK!
Testing toSexp/fromSexp ...OK!
- : unit = ()
```

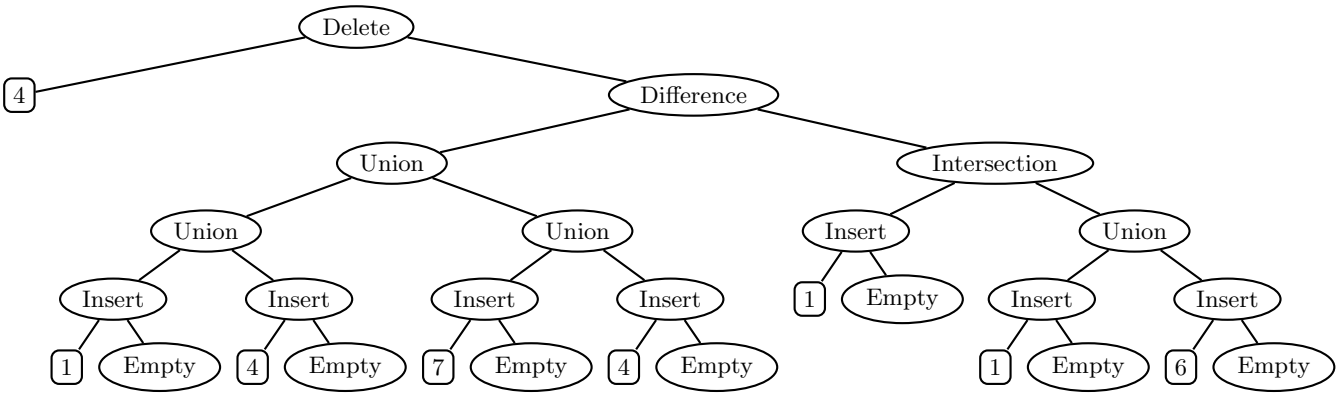
Figure 4: Sample use of `BSTSetTest.testSmall()`.

**Problem 2 [25]:** `OperationTreeSet`

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` create 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 very 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.

In this problem, you are asked to flesh out the missing operations in the skeleton of the `OperationTreeSet` module in Fig. 5. 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 fleshing out the definitions of the `member`, `toList`, `fromList`, `toSexp`, and `fromSexp` operations.

*Notes:*

- In `toList`, you may find it helpful to use functions in the `ListSetUtils` module. (These are also used in Handout #15 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 $\geq 2$ elements, you should split first split the list into two (nearly) equal sublists (using `alts` from PS1 & PS2, say) and union the results of turning the sublists into sets. This yields a height-balanced operation tree.

- In `toSexp`, you should represent each non-empty node in the operation tree as an sexpression 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)))
```

- You can perform a suite of automatic tests on your `OperationTreeSet` implementation by executing `OperationTreeSet.testSmall()` or `OperationTreeSet.testBig()`.

```
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 member x s = false (* Replace this stub *)

  let rec toList s = [] (* Replace this stub *)
    (* You may use operations in ListSetUtils, using the abbreviation LSU
       defined above. *)

  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 toSexp eltToSexp s = Sexp.Seq [] (* Replace this stub *)
    (* Returns an s-expression that shows the structure of the tree.
       See the PS3 description for examples *)

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

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

end
```

Figure 5: Skeleton of the `OperationTreeSet` module.

**Problem 3 [25]: 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 the previous two problems, we will use the somewhat different `PRED_SET` signature shown in Fig. 6. Here is a comparison of `PRED_SET` with `SET`:

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

```
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 6: 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 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
# member 100 s;;
- : bool = false
```

The set `ps1` 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. Te 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. [15]: `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/ps3/PredSet.ml`. Each of your definitions should be a one-liner. (For `fromList`, you may use operations from the `List` module.) Convince yourself that your implementation is correct by testing some simple examples.

**b. [10]: Other Functions**

In this problem, 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 -> 'a set`
   Returns `true` if all of the elements of the first set parameter are are also elements of the second set parameter, and `false` otherwise.

*Problem Set Header Page*
*Please make this the first page of your hardcopy submission.*

# CS251 Problem Set 3
## Due 6pm Saturday, February 21

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 your problem set.*

| **Part** | **Time** | **Score** |
|---|---|---|
| General Reading | | |
| Problem 1 [50] | | |
| Problem 2 [25] | | |
| Problem 3 [25] | | |
| **Total** | | |