CS251 Programming Languages **Handout # 34'**
Prof. Lyn Turbak **Thursday, April 4**
Wellesley College

Problem Set 6

Due: Friday, April 5, 2002

Note: This is the final version of PS6. Problems 1–4 are due Friday, April 5. Problems 5 is due Monday, April 8.

Reading:

- Handouts: Introduction to ML (#28), More ML (#23), Type Checking (#30), SMLNJ (#31), SML Exercises (#32), Polymorphic Types (#32)
- Paulson's ML for the Working Programmer (*MLWP*), Chapters 2, 3, 4, 5.1-5.11, 9.

Submission:

- Problems 1 and 2 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- For Problems 3, 4, and 5, your softcopy submission should include a copy of your entire `ps6` directory.
- Your hardcopy submission for Problem 3 should be the files `prob3a.hem` and `prob3b.hem`.
- Your hardcopy submission for Problem 4 should be the files `prob4a.hep` and `prob4b.hep`.
- Your hardcopy submission for Problem 5 should be the files `MergeSort.sml` and `TwoThreeTree.sml`

Problem 1 [20]: Type Derivations

Consider the following HOFL expression:

```
(((abs (a b)
    (abs (f)
      (f b a)))
  1 true)
 (abs (x y)
   (if x y 0)))
```

a. **[3]** Translate the above expression into an explicitly typed HOFLEMT expression by adding appropriate type annotations.

b. **[17]** Give a typing derivation that proves that the explicitly typed HOFLEMT expression from Part (a) is well-typed in an empty type environment.

As explained in Handout #30, a typing derivation is an upside-down tree in which every node is a type judgement of the form $A \vdash E : T$, where A is a type environment, E is an expression, and T is a type. Each node of the tree is the conclusion of the instantiation of one of the typing rules from from figure 4 of Handout #30; and the children of a node are the hypotheses of the rule instantiation.

Because the traditional tree-shaped (horizontal format) derivation would be *very* wide, you should use the vertical format for type derivations explained in Handout #30. You may abbreviate expressions and types to make your derivation more readable as long as you give explicit definitions for each abbreviation. Make sure to label each rule by its name.

Problem 2 [20]: Polymorphic Type Derivations

Consider the following HOFL abstraction:

```
(abs (fs xs)
  (map (abs (f) (map (abs (x) (f x))
                     xs))
       fs))
```

a. **[3]** Translate the above expression into an explicitly typed HOFLEPT expression by adding appropriate type annotations. You will need to use HOFLEPT's polymorphic features.

b. **[17]** Give a typing derivation that proves that the explicitly typed HOFLEPT expression from Part (a) is well-typed in the following type environment:

$$A_1 = \{\texttt{map} \mapsto \texttt{(forall (A B) (→ ((→ (A) B) (listof A)) (listof B)))}\}$$

Follow the same advice about drawing type derivations that was given in Problem 2. Note that each node of your tree should be the conclusion of the instantiation of one of the typing rules from from figure 4 of Handout #30 or from figure 1 of Handout #33, and the children of a node are the hypotheses of the rule instantiation.

Problem 3 [15]: Explicit Types

For each of the HOFL programs in Fig. 1 annotate the program with type information so that it becomes a well-typed HOFLEMT program. In cases where a single function is used at more than one type, you will need to make separate copies of the function for each type at which it is used.

You can find the programs in Fig. 1 in the files `prob3a.hfl`,`prob3b.hfl`, `prob3a.hem`, and `prob3b.hem` within the directory `~/cs251/ps6`. The `.hfl` extension is for HOFL programs and `.hem` is for HOFLEMT programs. You should annotate the `.hem` version of each file so that it is a well-typed HOFLEMT program.

To test your solutions, follow the directions for running SML in Appendix A. When you are running SML connected to `~/cs251/ps6`, first evaluate

```
use("loadProb3.sml");
```

(you only need to do this once per session) and then evaluate

```
testProb3();
```

You are additionally encouraged to experiment with the HOFLEMT type checker on additional files of your own choosing. To do this, evaluate

```
use("loadHoflemtTypeCheck.sml");
```

and then evaluate

```
TypeCheck.checkFile filename;
```

where filename is the name of a file containing the HOFLEMT program you wish to type check. Make sure the filename is relative to the current directory; e.g. `"foo.hfl"` must be in the current directory, `"test/bar.hfl"` must be in the subdirectory `test` of the current directory, and `"../baz.hfl"` must be in the parent directory of the current directory.

a. [5]

```
(program (a)
  (bindrec ((generate
              (abs (seed next done?)
                 (if (done? seed)
                     (empty)
                     (prepend seed
                       (generate (next seed) next done?)))))
            (foldr
              (abs (binop init xs)
                (if (empty? xs)
                    init
                    (binop (head xs)
                           (foldr binop init (tail xs))))))
           )
     (bind lst (generate a (abs (x) (- x 1)) (abs (y) (= y 0)))
       (if (foldr (abs (x y) (bor (> x 5) y)) false lst)
           (foldr (abs (x y) (+ x y)) 0 lst)
           (foldr (abs (x y) (* x y)) 1 lst)))))
```

b. [10]

```
(program (b)
  (bindpar ((inc (abs (x) (+ x 1)))
            (compose (abs (f g)
                       (abs (x) (f (g x)))))
            (thrice (abs (f)
                      (abs (x) (f (f (f x)))))))
     (bind nat (abs (g) ((g inc) b))
       (+ (nat (abs (h) (compose (thrice h) (thrice h))))
          (+ (nat (compose thrice thrice))
             (nat (thrice thrice)))))))
```

Figure 1: Annotate these HOFL programs to make them well-typed HOFLEMT programs.

Problem 4 [15]: Explicit Polymorphic Types

For each of the two HOFL programs in Fig. 1, annotate the program with type information so that it becomes a well-typed HOFLEPT program. Recall that HOFLEPT supports polymorphism via the `pabs` and `papp` expressions and `forall` type. Unlike in Problem 3 above, here it is not necessary to duplicate any function definitions. Rather, for function definitions that must be duplicated for HOFLEMT, HOFLEPT gives such functions polymorphic types and uses `papp` to instantiate the polymorphic type differently for different uses.

You can find the programs in Fig. 1 in the files `prob4a.hfl`,`prob4b.hfl`, `prob4a.hep`, and `prob4b.hep` within the directory `~/cs251/ps6`. The `.hfl` extension is for HOFL programs and `.hep` is for HOFLEPT programs. You should annotate the `.hep` version of each file so that it is a well-typed HOFLEPT program.

To test your solutions, first launch SML connected to `~/cs251/ps6`, and then evaluate (one time only)

```
use("loadProb4.sml");
```

and then evaluate

```
testProb4();
```

You are additionally encouraged to experiment with the HOFLEPT type checker on additional files of your own choosing. To do this, evaluate

```
use("loadHofleptTypeCheck.sml");
```

and then evaluate

```
TypeCheck.checkFile filename;
```

where filename is the name of a file containing the HOFLEPT program you wish to type check.

Problem 5 [30]: ML Programming

In this problem, you are asked to write some functions in SML. For examples of ML functions, see:

- Handouts #28 and #29.
- The on-line SML tutorials linked from the CS251 home page.
- The Paulson book, *ML for the Working Programmer*. I have left three copies of this book in the Linux lab for your convenience.

a. [14]: Merge Sort

In this problem, you are to implement a merge sort algorithm on lists by implementing the following functions in the file `MergeSort.sml` within the `ps6` directory:

- `val split : 'a list -> ('a list * 'a list)`
  `split` *xs* returns a pair of lists *ys* and *zs* such that (1) *ys* `@` *zs* is a permutation of *xs* and (2) `|` `length(`*ys*`) - length(`*zs*`) |` ≤ 1. For example, some possible results of `split [7,3,5,2,6]` are `([7,3,5], [2,6])`, `([6,2,5], [3,7])`, and `([7,5,6], [3,2])`.
- `val merge : ('a * 'a -> bool) -> 'a list -> 'a list -> 'a list`
  Assume that *xs* and *ys* are lists sorted according to the less-than-or-equal-to predicate *leq*. Then `merge` *leq xs ys* should return a list of the elements in *xs* and *ys* sorted according to *leq*.
- `val sort : ('a * 'a -> bool) -> 'a list -> 'a list`
  `sort` *leq xs* returns a list of the elements in *xs* sorted according to the less-than-or-equal-to predicate *leq*. Sorting should be performed by the merge sort algorithm, which requires splitting a list into two (almost) equal sublists, and merge the result of recursively sorting the sublists.

You can compile your program by executing the following in the `*sml*` buffer:

```
CM.make'("MergeSort.cm");
```

You should test your functions to make sure that they work correctly. The following testing functions have been provide for your convenience.

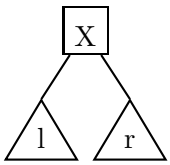
```
fun down 0 = []
  | down n = n::(down (n-1))

fun sortTest n = sort (fn (x,y) => (x < y)) (down n)
```

b. [16]: 2-3 Trees

Many data structures use some sort of balanced search tree to guarantee that various operations on the data structure are efficient. A particularly elegant form of balanced search tree is a **2-3 tree**. A 2-3 tree has three different kinds of nodes:

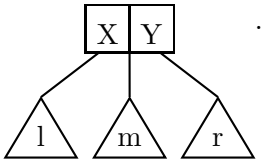
1. A leaf, written as ●.
2. A 2-node, written as



   X is called the **value** of the 2-node; l is its **left subtree**; and r is its **right subtree**. Every 2-node must satisfy the following invariants:

   (a) Every value v appearing in subtree l must be $\leq X$.
   (b) Every value v appearing in subtree r must be $> X$.
   (c) The length of the path from the root of the 2-node to *every* leaf in its subtrees must be the same.

3. A 3-node, written as



X is called the **left value** of the 3-node; Y is called the **right value** of the 3-node; l is its **left subtree**; m is its **middle subtree**; and r is its **right subtree**.

Every 3-node must satisfy the following invariants:

1. Every value v appearing in subtree l must be $\leq X$.
2. Every value v appearing in subtree m must be $> X$ and $\leq Y$.
3. Every value v appearing in subtree r must be $> Y$.
4. The length of the path from the root of the 3-node to *every* leaf in its subtrees must be the same.

The path-length invariant on 2-nodes and 3-nodes means that 2-3 trees are necessarily balanced; the height of a 2-3 tree with n nodes cannot exceed $\log_2(n+1)$. Together, the tree balance and the ordered nature of the nodes means that looking up information in a 2-3 tree takes logarithmic time. In this problem, we shall see that inserting a new element into a 2-3 tree also takes logarithmic time.

Given a collection of three or more values, there are several 2-3 trees containing those values. For instance, Fig. 2 shows the four distinct 2-3 trees containing first 7 positive integers.

Fig. 3 gives the definition of an SML datatype `TTTree` for 2-3 trees of integers and some associated operations. It is possible to extend the definition to any type of values supporting a comparison operator; see Extra Credit Problem 2.

The `elts` function returns a list of the elements of a 2-3 tree in sorted order. For example, `elts` returns `[1,2,3,4,5,6,7]` for each of the four 2-3 trees in Fig. 2. The `toString` and `printTree`

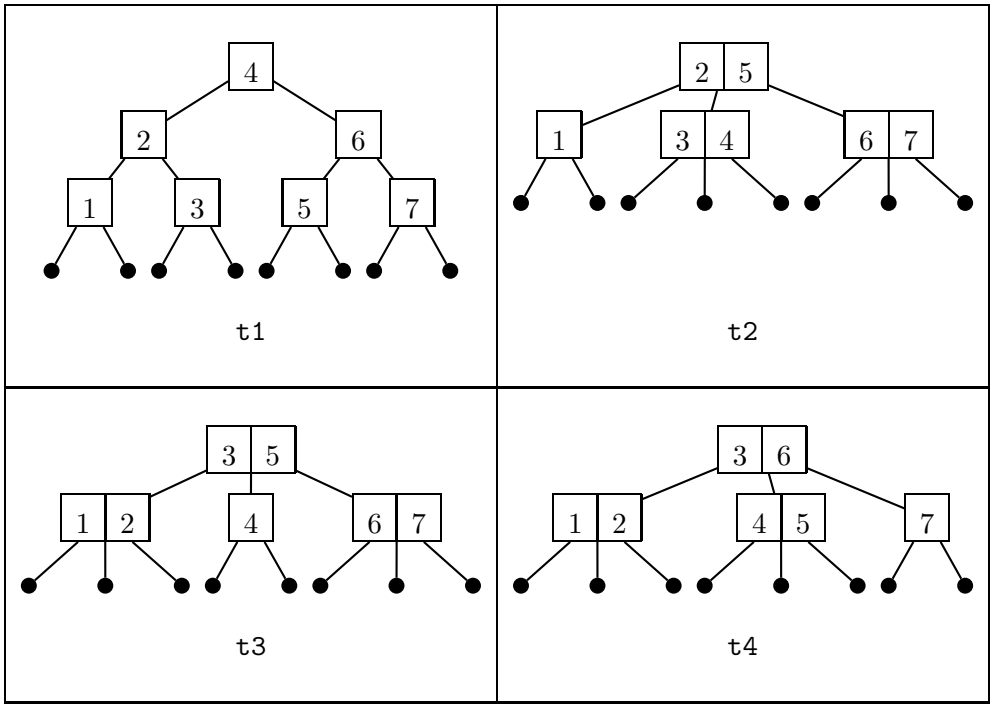


Figure 2: 2-3 trees containing the integers 1 through 7.

functions are use for displaying the contents of a 2-3 tree. The results of `printTree` for the four 2-3 trees in Fig. 2 are shown in Fig. 4.

```
(* SML datatype for 2-3 trees *)
datatype TTTree =
    Leaf
  | Two of TTTree * int * TTTree
  | Three of TTTree * int * TTTree * int * TTTree

(* val elts : TTTree -> int list *)
(* Returns the elements of a 2-3 tree in sorted order *)
fun elts Leaf = []
  | elts (Two(l,v,r)) = (elts l) @ [v] @ (elts r)
  | elts (Three(l,v1,m,v2,r)) =
      (elts l) @ [v1] @ (elts m) @ [v2] @ (elts r)

(* val toString : TTTree -> string *)
(* Returns a string representation of a 2-3 tree *)
fun toString tr =
    let fun strings Leaf = []
          | strings (Two(l,v,r)) =
              (stringsIndent r)
              @ [Int.toString(v)]
              @ (stringsIndent l)
          | strings (Three(l,v1,m,v2,r)) =
              (stringsIndent r)
              @ [Int.toString(v2)]
              @ (stringsIndent m)
              @ [Int.toString(v1)]
              @ (stringsIndent l)
        and stringsIndent t = (map (fn s => (""^ s)) (strings t))
     in foldr (fn (s1,s2) => s1 ^ "\n" ^ s2) "" (strings tr)
    end

(* val printTree : TTTree -> unit *)
(* Prints a textual representation of a 2-3 tree. *)
fun printTree t = (print (toString t); print "\n")
```

Figure 3: An SML datatype for 2-3 trees of integers and some associated operations.

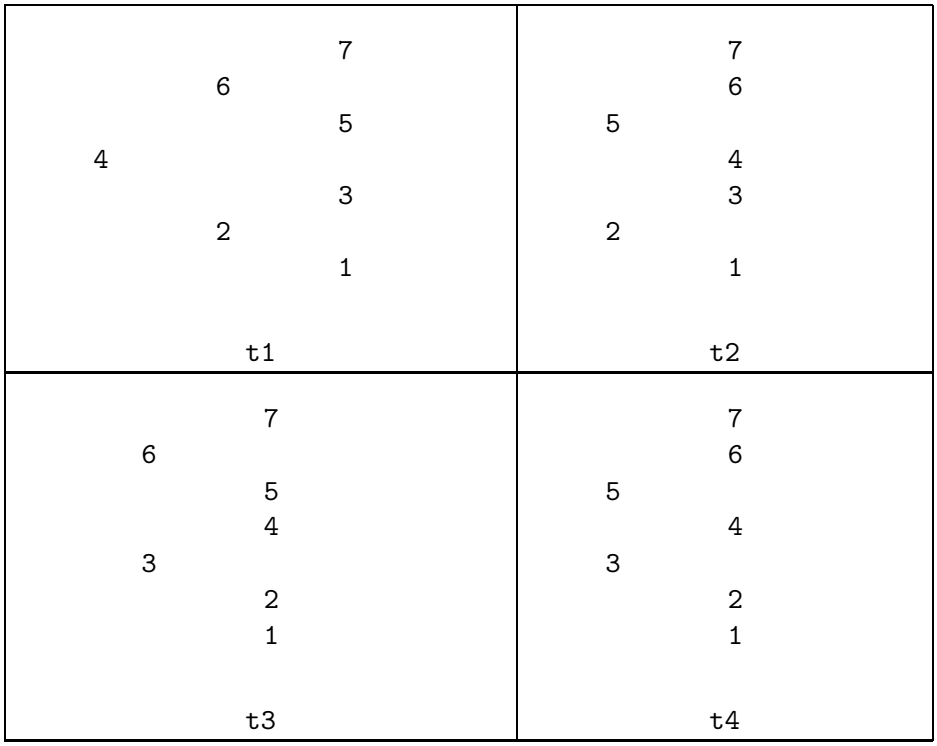


Figure 4: Results of using `printTree` to display the contents of the four sample 2-3 trees. Turn the page 90 degrees clockwise to see the correspondence between the printed representations and the trees.

When inserting an element v into a 2-3 tree, care is required to maintain the invariants of 2-nodes and 3-nodes. As shown in Fig. 5, the order invariants are maintained much as in a binary search tree by comparing v to the node values encountered when descending the tree and moving in a direction that satisfies the order invariants. If v reaches a terminal 2-node (i.e., one with two leaf children) with value X, the balance invariant can be maintained by absorbing the inserted value v in a newly constructed 3-node with values v and X, as shown in Fig. 6.

However, if If v reaches a terminal 3-node with values X and Y, the value v cannot be absorbed. But it is possible to split the three values v, X, and Y into two terminal 2-nodes and a third value that is "kicked upstairs" because there is no room "downstairs" (see Fig. 7). As shown in Fig. 8, if there is a 2-node upstairs, the value kicked upward (call it w) can be absorbed by the 2-node. But if there is a 3-node upstairs, it cannot simply be absorbed, and another split takes place in which another value is kicked upstairs. This process continues until either the value w is absorbed or the root of the tree is reached (in which case a new 2-node root is created, increasing height of the tree).

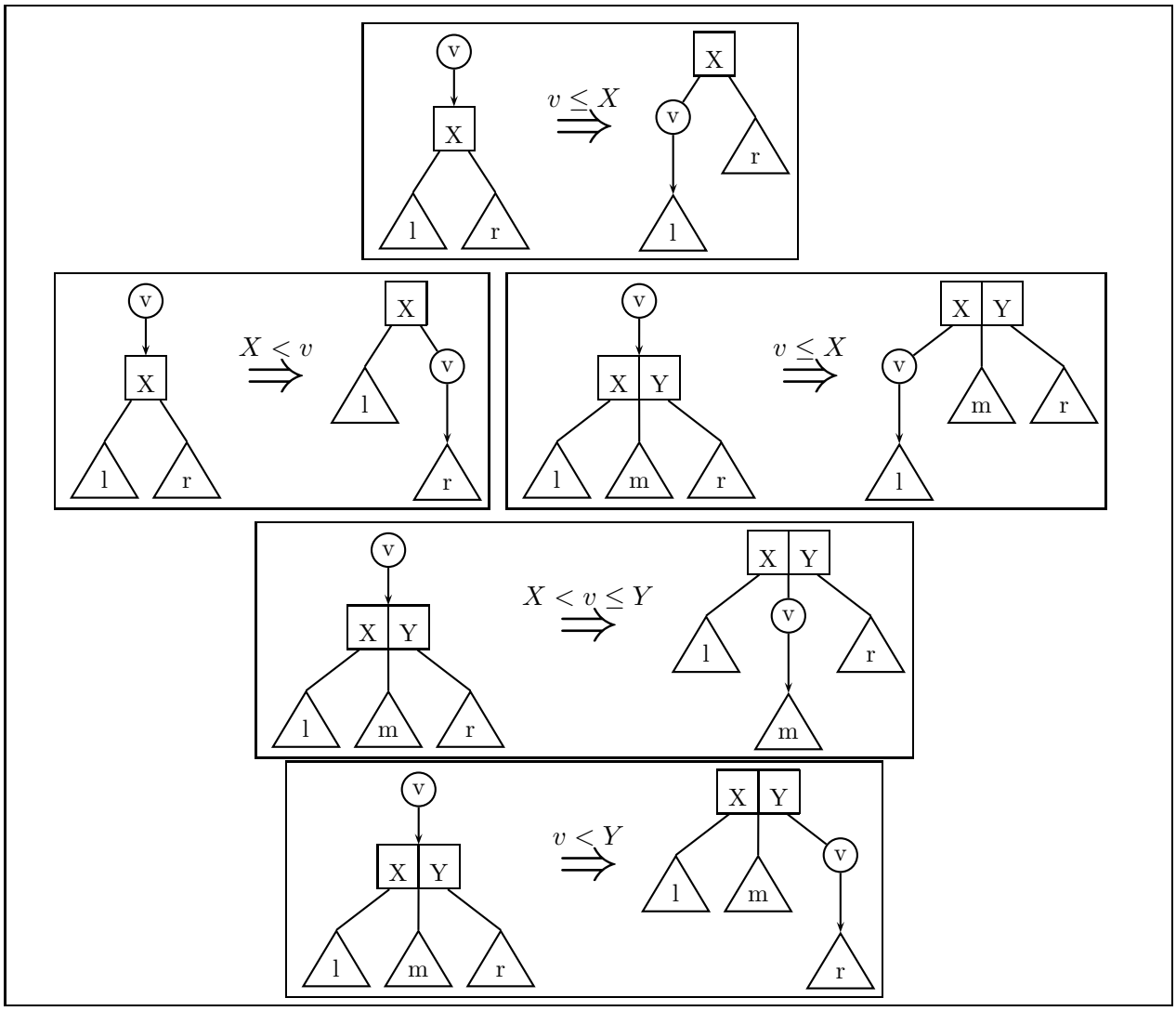


Figure 5: Rules for descending into a 2-3 tree when inserting a value v.

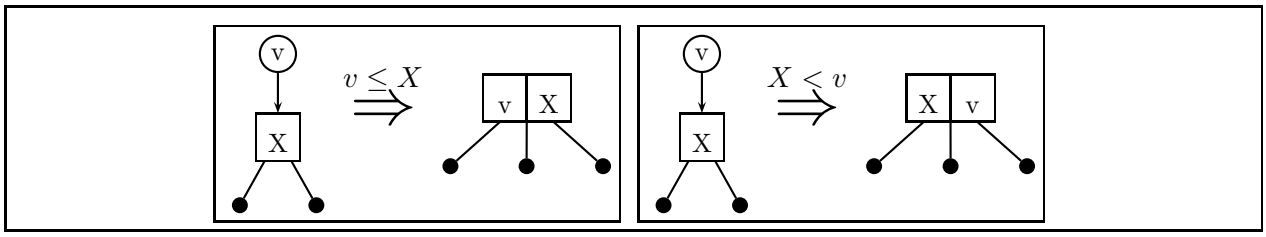


Figure 6: A value v inserted into a 2-node is absorbed.

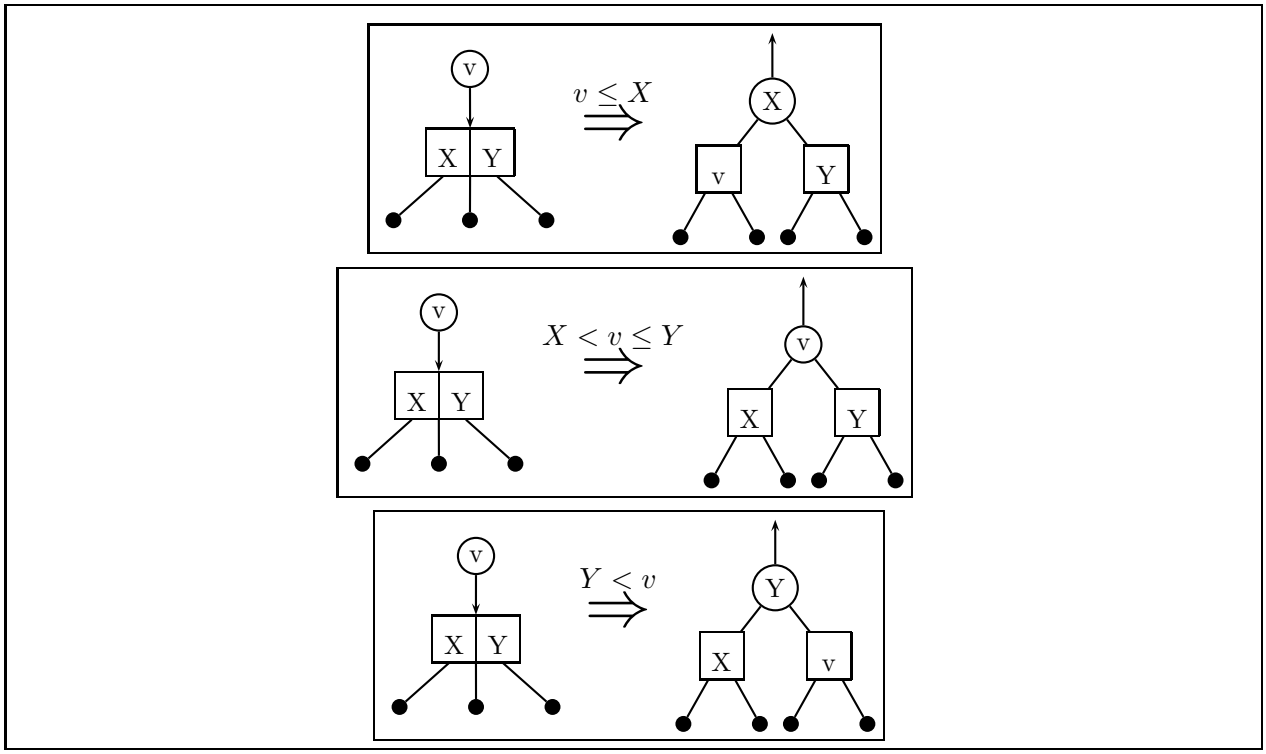


Figure 7: A value v inserted into a 3-node causes a split in which one value is "kicked upstairs".

Your goal in this part is to implement the 2-3 tree insertion in SML by writing an `insert` function with the following signature:

```
val insert : (int * TTTree) -> TTTree
```

You should define `insert` in terms of an auxiliary function `ins` that implements the rewrite rules shown in the figures above. The `ins` function should have the following signature:

```
val ins : int -> TTTree -> TTInsertResult
```

where `TTInsertResult` is defined as follows:

```
datatype TTInsertResult =
    OK of TTTree
  | Up of TTTree * int * TTTree
```

A result `OK(t)` indicates the inserted value was absorbed an that a new tree `t` was created inthe process of absorbing the inserted value. A result tagged `Up(l,w,r)` indicates that insertion has resulted in a value `w` being "kicked upstairs" with left subtree `l` and right subtree `r`.

To do this problem, you should flesh out the skleton of `insert` and `ins` provided in the file `TwoThreeTree.sml` within the `ps6` folder. You can compile the file by executing the following in the `*sml*` buffer:

```
CM.make'("TwoThreeTree.cm");
```

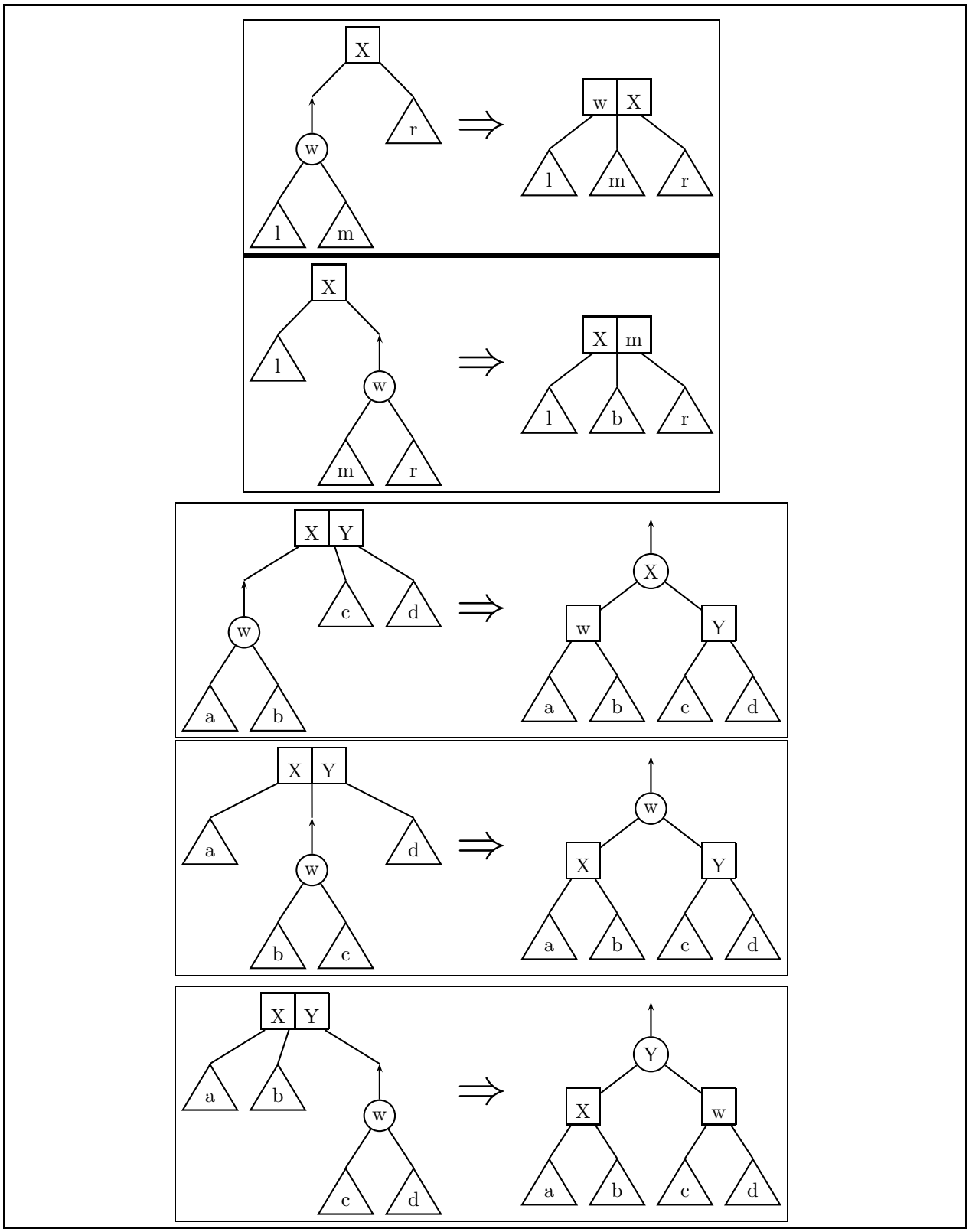


Figure 8: Rules for handling a value "kicked upstairs".

You should test your `insert` function to make sure it works appropriately. The following functions, in conjunction with the `printTree` function described above, are helpful for testing. These have also been provided in the file.

```
fun listToTree xs = (List.foldr insert Leaf xs)

fun ttsort xs = elts (listToTree xs)

fun down 0 = []
  | down n = n::(down (n-1))

fun ttsortTest n = ttsort (down n)
```

For instance, `printTree(listToTree(down(7)))` should print out one of the tree representations in Fig. 4 – which one?

Note that 2-3 trees suggest an straightfoward $n \cdot \log(n)$ list sorting algorithm: insert all elements from the list into a 2-3 tree and then return the sorted elements via `elts`. This idea is implemented via the `ttsort` function above.

Extra Credit 1 [20]: ML Types

Figs. 9–10 contain twenty higher-order ML functions. For each function, write down the type that would be inferred for it in SML. For example, consider the following SML `length` function:

```
fun length [] = 0
  | length (_::xs) = 1 + (length xs)
```

The ML type of this function is:

```
val length : 'a list -> int
```

Note: you can check your answers by typing them into the ML interpreter. But please write down the answers first before you check them — otherwise you will not learn anything!

```
fun id x = x

fun compose f g x = (f (g x))

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

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

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

fun churchPair x y f = f x y

fun generate seed next done =
      if (done seed) then []
      else seed :: (generate (next seed) next done)

fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)

fun filter pred [] = []
  | filter pred (x::xs) =
      if (pred x) then x::(filter pred xs)
      else (filter pred xs)

fun product fs xs =
      map (fn f => map (fn x => (f x)) xs) fs
```

Figure 9: A sampler of higher-order functions in ML, part 2.

```
fun zip ([], _) = []
  | zip (_, []) = []
  | zip (x::xs, y::ys) = (x,y)::(zip(xs,ys))

fun unzip [] = ([], [])
  | unzip ((x,y)::xys) =
      let val (xs,ys) = unzip xys
       in (x::xs, y::ys)
      end

fun foldr binop init [] = init
  | foldr binop init (x::xs) =
      binop(x, foldr binop init xs)

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

fun flatten lst = foldr op@ [] lst

fun forall pred [] = true
  | forall pred (x::xs) =
      pred(x) andalso (forall pred xs)

fun exists pred [] = false
    | exists pred (x::xs) = (pred x) orelse (exists pred xs)

fun some pred [] = NONE
  | some pred (x::xs) = if (pred x) then SOME x else some pred xs

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

fun some2 pred = oneListOpToTwoListOp some pred
```

Figure 10: A sampler of higher-order functions in ML, part 2.

Extra Credit 2 [10]: Generalizing 2-3 Trees

Generalize the 2-3 tree implementation from Problem 5 to handle any tree component of type `'a` that is equipped with a `compare` method having the following signature:

```
val compare : ('a * 'a) -> order
```

Recall that `order` is a built-in datatype with the definition:

```
datatype order = LESS | EQUAL | GREATER
```

Extra Credit 3 [30]: Implementing 2-3 Trees in Java/C/C++

Implement 2-3 trees in one of Java, C, or C++. You may choose to make your trees immutable (as in Problem 5); alternatively, you may make them mutable. Your implementation should support the operations `insert`, `elts`, `toString`, and `printTree`. Additionally, you should provide some way to return an empty tree.

A Using SML

Here are the steps you need to follow to use SML:

1. In a shell in the `~/cs251` directory, perform a `cvs update -d` to grab all relevant files.
2. Start SMLNJ within Emacs by typing `M-x sml` ENTER.
3. Go to the SMLNJ interpreter buffer via `C-x b *sml*` ENTER.
4. In Emacs, change the default directory used by `sml` by typing `M-x sml-cd` ENTER *dir*, where *dir* is the name of the directory you wish to be the default directory for finding SML files. For this problem set, you *dir* to be `~/cs251/ps6`.
5. Compile and load the subsystem you are interested in running. The `ps6` directory contains several "load" files and Configuration Manager (.cm files) for loading and compiling the various subsystems you might want to run in this assignment. You load one of the load files by using SML's `use` command, and compile on of the .cm files by using SML's `CM.make'` command. Here's how you load and compile the specific subsystems for this assignment:

   - For testing problem 3: `use("loadProb3.sml");`
   - For testing problem 4: `use("loadProb4.sml");`
   - For testing problem 5a: `CM.make'("MergeSort.cm");`
   - For testing problem 5b: `CM.make'("TwoThreeTree.cm");`
   - For experimenting with the HOFL evaluator: `use("loadHoflEval.sml");`
   - For experimenting with the HOFLEMT type checker: `use("loadHoflemtTypeCheck.sml");`
   - For experimenting with the HOFLEPT type checker: `use("loadHofleptTypeCheck.sml");`

   Executing the above expressions will cause many lines of text to appear on the screen. Although some of the lines seem to indicate some sort of error, you can ignore these. Here's an example of something you can safely ignore:

   ```
   [checking ../sml/hoflemt/CM/x86-unix/Pretty.cm.stable ... not usable]
   ```

   You know that everything has compiled OK if the lines of text generated after `use` or `CM.make'` ends with the following:

   ```
   val it = () : unit
   ```

6. Run the desired SML function:

   - For testing problem 3: `testProb3();`
   - For testing problem 4: `testProb4();`
   - For testing problem 5a: `MergeSort.msortTest(`*n*`)`, where *n* is an integer.
   - For testing problem 5b: `TwoThreeTree.insertTest(`*n*`)`, where *n* is an integer.

- For experimenting with the HOFL evaluator: `Eval.runFile` *`filename args`*, where *filename* is a string naming the file in which the HOFL program resides and *args* is an SML list of integer arguments for the program. Make sure the filename is relative to the current directory; e.g. `"foo.hfl"` must be in the current directory, `"test/bar.hfl"` must be in the subdirectory `test` of the current directory, and `"../baz.hfl"` must be in the parent directory of the current directory.
- For experimenting with the HOFLEMT type checker: `TypeCheck.checkFile` *`filename`*, where *filename* is a string naming the file in which the HOFLEMT program resides.
- For experimenting with the HOFLEPT type checker: `TypeCheck.checkFile` *`filename`*, where *filename* is a string naming the file in which the HOFLEPT program resides.

In the some of the cases, you can reduce the amount of typing by "opening" SML structures. E.g., if you execute

```
open TypeCheck;
```

this makes all the components of the `TypeCheck` structure available without having to prefix them with `"TypeCheck."`. For example, after executing the above line, you can then execute `checkFile` *`filename`*. However, you must be careful to re-perform the `open` command if you ever recompile the file containing the structure! If you do not do this, the unprefixed names will refer to the old version, not the new version.

Handy Tidbits:

- In the SML intepreter, typing `M-p` cycles backwards through previous expressions typed at the interpreter, and typing `M-n` cycles forwards. Use these often to avoid unnecessary typing!
- If you see printed entities in angle brackets, such as `<Sexpr>`, or ellipses (`...`), then your **print depth** may be too low. You can reset it to a number n as follows:

```
Compiler.Control.Print.printDepth := n
```

A good value for n is 1000.

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

CS251 Problem Set 6
Due Friday, April 5

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

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 [20]		
Problem 2 [20]		
Problem 3 [15]		
Problem 4 [15]		
Problem 5 [30]		
Extra Credit 1 [20]		
Extra Credit 2 [10]		
Extra Credit 3 [30]		
Total		