Computer Science 131, Fall 2001
Assignment 1: SML Warmup

Out: Thursday, September 6

Due: Thursday, September 13, 2:45pm

This assignment consists of three problems, each of which requires defining two or more functions. You should get the file `assign1.sml` from the CS 131 web pages; go to

```
http://www.cs.hmc.edu/courses/current/cs131
```

and follow the "Schedule, Slides, and Assignments" link. Modify this file to include your solutions.

The submission process should be familiar; when you are ready, run the command

```
cs131submit assign1.sml
```

Your final submission *must not have syntax or type errors*. If you know the code does not work, explain why in a comment; code that does not even compile should be commented out. Functions must have the exact names and types given in the problem below (though you are free to define other helper functions as well), and should be reasonably well-commented to help the graders understand your code. Comments in ML are written as:

```
(* ... comment goes here... *)
```

This assignment uses material covered in class, which appears in more detail in Chapters 1, 2, 3, and 6 of the course textbook.

1 Defining Functions (20%)

1. Define the function

   ```
   fib : int -> int
   ```

   which computes the Fibonacci number F_n when given $n \geq 0$. The Fibonacci numbers are defined by:

   $$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{when } n \geq 2. \end{aligned}$$

Your code need not be efficient; the most obvious definition turns out to take time exponential in n. You need not check for negative inputs.

2. It is occasionally useful to consider a more general specification:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \textbf{for all integers } n. \end{aligned}$$

   This uniquely determines the value of F_n for all integer n, not just the non-negative integers. (For example, what must F_{-1} be to satisfy this specification? What does this mean for F_{-2}?) Define the function

```
intfib : int -> int
```

   that computes the value of F_n given an arbitrary integer n. Again, efficiency is secondary.

2 Red-Black Trees (20%)

[Although the description of this problem is lengthy, the code you need to write is fairly short.]

In class you saw code to implement ordered binary trees containing integers, and the corresponding `member` and `insert` functions which use trees to represent sets of integers. For code that repeatedly checks whether an integer is in a given set or not, this tree structure can be much more efficient than using a list of integers; for well-balanced trees the running time of `member` would be proportional to the log of the size of the set.

However, there is no guarantee that the trees created by the insert function will be well-balanced. A very unbalanced tree results, for example, if one inserts a large number of integers in increasing or in decreasing order. When the tree is this unbalanced, the time to search such a tree degrades to linear in the number of nodes.

To fix this problem, a number of more complicated data structures have been proposed which prevents unbalanced trees from being constructed by insert. In this problem we will look at a data structure called "Red-Black Trees".

The main idea is fairly simple. We will again use ordered binary trees as before, but we will assign each node in the tree a color, red or black. Thus our SML representation for these trees will be:

```
datatype color = R | B
datatype rbtree = Empty
                | Node of color * rbtree * int * rbtree
```

However, we will permit only certain colorings of the nodes. We say that a red-black tree is well-formed if it satisfies the following two properties:

1. No red node has a red parent.

2. Every path from the root to an empty node contains the same number of black nodes.

(Empty nodes will be considered black for the purposes of these properties.)

If a tree satisfies these properties than it has to be at least reasonably-well balanced. The longest possible path the root to an empty node, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only. It can be shown that this guarantees $O(\log n)$ searches if the tree contains n nodes.

Because this will still be an ordered tree, the lookup function is nearly unchanged; the colors don't matter, so we just add a wildcard pattern.

```
fun member (n, Empty) = false
  | member (n, Node(_,left,m,right)) =
     if (n = m) then
        true
     else if (n < m) then
        member(n, left)
     else
        member(n, right)
```

The hard part is in inserting an element into a well-formed tree, while maintaining the two color invariants. We will do this essentially as before, but with the three modifications.

- We will try to insert the new node as a red node. Inserting a new red node into the tree cannot cause a violation of the second invariant, but might create two consecutive red nodes. (We cannot have more than two consecutive red nodes because we started out with a well-formed tree.)

- Therefore, after each recursive insertion we will look to see whether we obtained any of the four "bad" trees shown in Figure 1. (In this figure, the letters `x`, `y`, and `z` represent arbitrary integers in non-`Empty` nodes, while `a` through `d` represent arbitrary subtrees.) If so, we will get rid of a red-red violation by turning this into the "good" tree in the middle of that figure. These four transformations get rid of a red-red violation without changing the number of black nodes in the paths of the tree, so we still do not introduce any violations of the second invariant. Also, these transformations preserve the ordered property of the tree: the integers in the left subtree are less than the contents of the node, which in turn is less than all the integers in the right subtree.

- The four bad-tree patterns in Figure 1 will detect a red-red violation when we get around to looking at the parent of the top red node. But what if the red-red violation is the root of whole tree and one of its children? We can fix this easily by recoloring the root of the tree black. This gets rid of the red-red violation and adds one black node to *every* path from the root to a leaf. If all such paths had equal numbers of black nodes before, they'll still have equal numbers of black nodes.

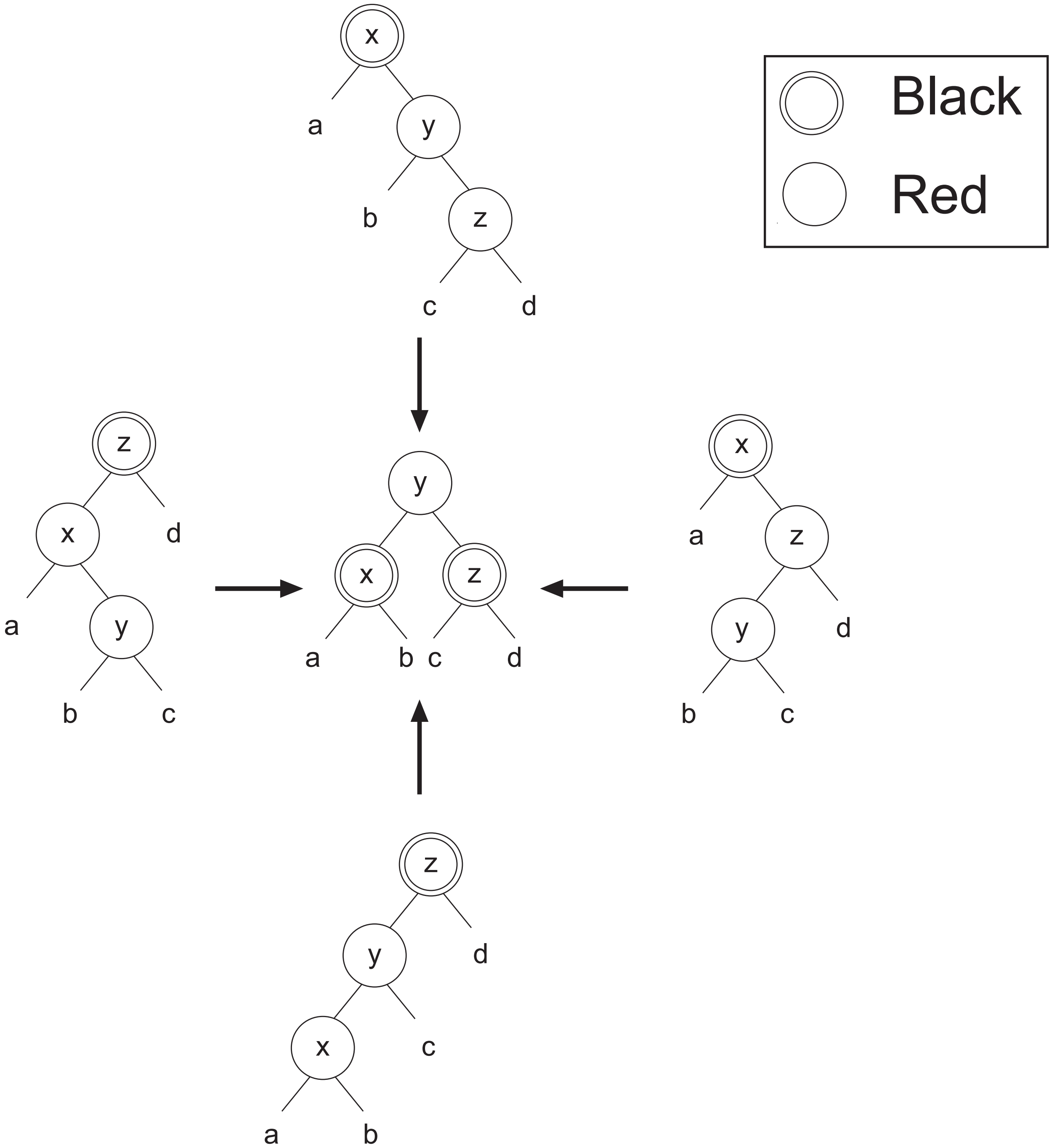

Figure 1: Four bad trees and how to fix them up

You have been given most of the code for the `insert` function; the only part remaining to be implemented is `balance`, which checks whether its argument is one of the four bad trees in Figure 1 and if so returns the corresponding good tree instead. For all other inputs, the `balance` function should return its argument unchanged.

[When you're done, it is interesting to compare the code for this insert function with the code found in typical algorithms texts like CLR; the SML code is much more elegant and easier to understand.]

3 Arithmetic Expressions (60%)

Consider the following datatype for representing arithmetic expressions.

```
datatype opn = Add | Sub | Mult | Divide
datatype aexp = Num  of real
              | Opn of aexp * opn * aexp
```

Then `3.0 + (9.0 / 2.5)` is represented as

```
Opn(Num 3.0, Add, Opn(Num 9.0, Divide, Num 2.5))
```

Certain HP calculators and certain programming languages evaluate expressions using a stack; here an arithmetic computation is expressed as a sequence of operations, which will be represented with the following datatype:

```
datatype sopn = Push of real
              | DoOpn of opn
              | Swap
```

The operation `Push r` means push the number r onto the stack; the operations `DoOpn Add`, `DoOpn Sub`, `DoOpn Mult`, and `DoOpn Divide` mean to replace the top two numbers on their stack with their sum, difference, etc.; the `Swap` operation swaps the top two numbers on the stack:

If the stack looks like:	and the operation is:	afterwards the stack should be:
$\ldots$	`Push` r	$r \ldots$
$a\ b \ldots$	`DoOpn Add`	$(b + a) \ldots$
$a\ b \ldots$	`DoOpn Sub`	$(b - a) \ldots$
$a\ b \ldots$	`DoOpn Mult`	$(b * a) \ldots$
$a\ b \ldots$	`DoOpn Divide`	$(b/a) \ldots$
$a\ b \ldots$	`Swap`	$b\ a \ldots$

(Note: the top of the stack is shown on the left.)

1. We will represent the stack as a list

   ```
   type stack = real list
   ```

with the front of the list corresponding to the top of the stack.

Write a recursive function

```
evalRPN : sopn list * stack -> real
```

which returns the number at the top of the stack after performing the given operations in order, starting with the given stack. Be careful to get the order right for `Sub` and `Divide`;

```
evalRPN ([Push 2.0, Push 1.0, DoOpn Sub], [])
```

should return `1.0`, not `~1.0`.

At this point in the course you need not worry about stack underflow, numeric overflow, or divide-by-zero. Correct code may therefore generate non-exhaustive match warnings.

2. Write a function

```
toRPN : aexp -> sopn list
```

which converts an arithmetic expression to a list of stack operation instructions which compute the same expression. There should be an `DoOpn Add` stack operation for every `Add` in the input, and so on; evaluating the input expression to a number r and then returning `[Push` r`]` is not acceptable.

Hint: this corresponds exactly to a postfix traversal of the input expression viewed as a tree.

3. The same expression can be computed several ways in the stack machine, because for each subexpression you can choose to evaluate the left side first or the right side first. (Because subtraction and division are not commutative, evaluating the right side first and then the left will require a `Swap` to fix things up.)

For example, $1.0 - (2.0 + 3.0)$ can be computed either by the sequence

```
[Push 1.0, Push 2.0, Push 3.0, DoOpn Add, DoOpn Sub]
```

or by

```
[Push 2.0, Push 3.0, DoOpn Add, Push 1.0, Swap, DoOpn Sub]
```

Note that the first set of instructions requires that the stack be able to hold at least three numbers simultaneously, while the second sequence never requires more than two numbers on the stack at any one time.

Define the function

```
toRPNopt : aexp -> sopn list * int
```

which returns a pair containing (1) an optimal sequence of operations to evaluate the given arithmetic expression, and (2) the maximum number of values simultaneously on the stack during the execution of this sequence. Optimal here is defined to mean having the least possible maximum number of values on the stack at any given time. (In other words, you are trying to minimize the maximum stack depth during evaluation.)

Hints: (1) This function can be computed inductively, using the optimal instruction sequence for the first operand, the optimal sequence for the second operand, and the stack sizes they each require. (2) You can decide whether to evaluate the left side first or the right side first just by looking at the stack depth each requires (and without looking at the particular operations).

4. **Extra Credit (20%)** Write the function

```
fromRPN : sopn list -> aexp
```

that converts a list of stack operations to the corresponding arithmetic expression. You may assume that evaluation of these stack operations starting with an empty stack would yield a stack containing only a single number, the answer. (That is, that the stack operations really do correspond to some arithmetic expression.)