CS109A Assignment #7
Due 3:15 PM, 3/13/96

Please remember to separate the written homework from the programming problems and hand in each separately.

Written Homework

H1. (10 points) Place the words of this sentence in a binary search tree.

H2. (10 points) A *strict binary tree* is a binary tree with at least one node, in which every node has either both subtrees empty or neither subtree empty. Prove by structural induction that every strict binary tree has an odd number of nodes. Note: while we usually do structural inductions on binary trees starting with the empty tree as the basis, in this case the appropriate basis is a binary tree with a single node.

H3. (15 points) What is the probability that the integers 1 through 15, placed at random into an array A of length 15, will form a partially ordered tree, i.e., $A[i/2] > A[i]$ for $i = 2, 3, \ldots, 15$? (Special case of Exercise 5.9.8, FCS) *Hint*: You might think that each of the 14 tests is either true or false with probability 1/2, so the probability of all 14 being true is 2^{-14}. However, the tests are not independent, so the product rule does not apply. You need to count the number of orders (that's easy: 15!) and then count the number of orders that form a heap. For example, you know that $A[1]$ must be 15. You also know that whatever 7 integers are in the left subtree, $A[2]$ must hold the largest of them, and so on. Happy counting!

H4. (15 points) We can represent a queue by an array, provided that the queue does not grow too long. In order to make the operations take $O(1)$ time, we must think of the array as *circular*. That is, the array `A[0..n-1]` can be thought of as having `A[1]` follow `A[0]`, `A[2]` follow `A[1]`, and so on, up to `A[n-1]` following `A[n-2]`, but also `A[0]` follows `A[n-1]`. The queue is represented by a pair of integers `front` and `rear` that indicate the positions of the front and rear elements of the queue. An empty queue is represented by having *front* be the position that follows *rear* in the circular sense; for example, $front = 23$ and $rear = 22$, or $front = 0$ and $rear = n - 1$. Note that therefore, the queue cannot have n elements, or that condition too would be represented by *front* immediately following *rear*. Thus the queue is full when it has $n - 1$ elements, not when it has n elements. Write functions in C for the queue operations *enqueue* and *dequeue*, assuming the circular array data structure. Do not forget to check for full and empty queues. (Part of Exercise 6.8.3, FCS)

Programming Problems

For this assignment, you *must* make a copy of your code in the dropoff directory. You should call your file (or directory if you want to hand in several files) by your first initial, your last name, and the number 7. For example, John Robert Smith would copy his code to the file `/usr/class/cs109a/dropoff/jsmith7`. Remember you can't modify or retrieve what you copy, and if you feel the need to change it at the last minute, copy another file with another name, and tell us what you have done when you hand in the hardcopy. Under any circumstances, you still need to hand in your programs and scripts in hardcopy as usual.

This project will again involve the datatype from Project 6:

```
datatype ('operand) le =
    And of ('operand le) * ('operand le) |
    Or of ('operand le) * ('operand le) |
    Not of ('operand le) |
    Implies of ('operand le) * ('operand le) |
    Op of 'operand;
```

You may type this definition or download it from the class web page.

P1. (10 points) Write a function `noImplies(e)` that takes a value ("expression") e of datatype `le` and returns an equivalent expression that has no `Implies` operators. Show your code and a script of your function applied to the expression `exp` that is defined by:

```
val exp = And(Op("x"),Or(Not(Op("y")),Implies(Op("x"),Op("y"))));
```

Hint: Remember the equivalence we covered at the beginning of the course: $(x \rightarrow y) \equiv (\bar{x} + y)$.

P2. (15 points) The following function `insert(x,L)` produces list L if x is already on L and produces $L@[x]$ if x is not on L.

```
fun insert(x,nil) = [x]
|   insert(x, y::ys) =
        if x=y then y::ys
        else y::(insert(x,ys));
```

You may use this function to write a function `findVars(e)` that takes an expression e of datatype `le` and returns a list of the variables (strings) appearing in e. No string may appear more than once on the returned list, regardless of how many times it appears as an operand of e. Show your code and a script of your function applied to the expression `exp` of (P1). *Hint*: You might find it easier to write a "helper" function `fv1(e,L)` that takes an expression e and a list L of variables "already found" and returns a list containing all those variables on L plus any others that are found among the operands of e. Then call `fv1(e,nil)` to compute `findVars(e)`.

P3. (25 points) Write a function `tautology(e)` that takes an expression e of datatype `le` and tells whether e is true for every truth assignment for the variables of e. Show your code and a script of it running on the expression `exp` of (P1) and the expression `Or(Op("x"), Not(Op("x")))`. *Hints*:

1. You should use `findVars` of (P2) to get a list of the variables of the given expression e.

2. Using that list, you can write a function that cycles through each of the possible truth assignments. We can think of `false` as 0 and `true` as 1, so a truth assignment with n variables can be thought of as an n-bit binary integer. Cycling through the truth assignments is like starting with $00 \cdots 0$, and repeatedly adding 1 until we get a carry out of the nth place, at which time we know we have seen all n-bit integers.

3. Use function `eval` from Assignment 6 to evaluate e for each truth assignment.

4. While we have stressed the use of exceptions to catch unexpected errors, you can also create an exception that gets raised if we find a truth assignment that evaluates to false. This exception can be "handled" by a calling function that returns the value `false` if this exception is raised and returns `true` if we cycle through all the truth assignments (which can be indicated by another exception). Note that you can handle several exceptions, say `Foo` and `Bar` by an expression of the form

```
<function call> handle Foo => <desired result>
         | Bar => <desired result>
```