CS 538

Final Exam

Saturday, December 19, 1998

12:25 PM— 2:25 PM

Instructions

Answer any *four* questions. (If you answer more, only the first four will count.) Each question is worth 25 points. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (a) (15 points)

   Write Prolog facts and rules that define the relation `append3(L1,L2,L3,L4)`. `append3` represents a 3-way list append. That is, `append3(L1,L2,L3,L4)` is true if lists `L1`, `L2` and `L3` can be appended together to form list `L4`. For example, `append3([1,2],[3],[4,5],[1,2,3,4,5])` and `append3([1],[],[2],[1,2])` are true, while `append3([1],[2],[1],[1,2])` is false.

   (b) (5 points)

   Explain how Prolog would solve the query `append3([1],[2],[3],L)` using your definition of `append3`.

   (c) (5 points)

   Would your definition of `append3` work correctly for the query `append3(L1, L2,L3,[1,2,3])`? Why?

2. (25 points)

   Explain how type-inference works in ML. Illustrate the process by solving for the type of the following function

```
fun fm([],[],[]) = []
  | fm(h1::t1,h2::t2,h3::t3) =
      (if (h1) then (h2 h3) else h3) :: fm(t1,t2,t3);
```

   What does this function do?

3. (a) (8 points)

   Write an ML datatype definition that defines a polymorphic **trinary tree**, `'a triTree`. A trinary tree is defined to be either a **null** tree, or a **leaf** containing a single value of type `'a`, or a **node** that contains three subtrees of type `'a triTree`. Your definition should start

   ```
   datatype 'a triTree = (* and you fill in the rest *)
   ```

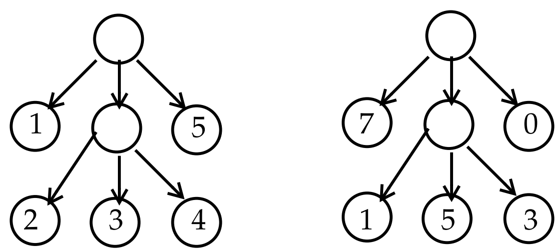
   (b) (8 points)

   Write an ML function `count`, of type `'a triTree -> int`, that counts the number of leaves in a trinary tree.

   (c) (9 points)

   Let `lessEq` be a function of type `('a * 'a) -> bool` that determines if one `'a` value is less than or equal another `'a` value.

   A trinary tree is said to be **ordered** if its leaf values, when visited in postorder (depth-first, left-to-right order) satisfy the `lessEq` relation. For example, the `int triTree` shown below on the left is ordered because $1 \le 2$, $2 \le 3$, etc. However, the `int triTree` on the right is not ordered since $7 > 1$, $5 > 3$, etc.



   Write an ML function `isOrdered(tree,lessEq)` of type

   ```
   'a triTree * ('a * 'a -> bool) -> bool
   ```

   that takes a `triTree` and an ordering relation, and determines if the `triTree` is ordered.

4. (a) (10 points)

   Write a Java class `Sort` that contains a member function

   ```
   void sort(int in[])
   ```

   that sorts an integer array, `in`, into sorted order. (You may use any sorting algorithm you wish).

   (b) (15 points)

   Now you are to use the features of Pizza to make `Sort` (and `sort`) **polymorphic**. This involves two things. You must parameterize class `Sort` so that it can sort an array of any type. Secondly, since you must compare values to do a sort, you'll need to add a comparison function, `greaterThan`, as a parameter to `sort`. Now sort is called as

   ```
   sort(valArray, greaterThan)
   ```

   `valArray` is an array of values to be sorted. `greaterThan` is a `boolean` function that determines whether a pair of array values are in the "greater than" relation.

5. (25 points)

   We have seen throughout the semester that **memoization** is a very useful optimization. We normally add memoization by hand, recoding an existing function. However, with a functional language like ML we can do better—we can **automate** the memoization process. Write an ML function `memoize(f)` that takes an arbitrary function `f` (of type `'a -> 'b`) and returns a function identical to `f` except that memoization is included. That is, whenever the memoized function is called with an argument `a` it has already seen, the value of `f(a)` previously computed is returned without being recomputed. If the memoized function is called with an argument, `a'` it has never seen, `f(a')` **is** computed, but it is also stored within the function so that the answer can be reused if the function is called again with `a'`.

6. (15 points)

   Let us consider a 3 by 3 square, containing values `V1` to `V9` as shown below.

V1	V2	V3
V4	V5	V6
V7	V8	V9

   These nine values can be conveniently represented in a list of the form `[V1,V2,V3,V4,V5,V6,V7,V8,V9]`. A square is said to be "semi-magic" if the values in each of the three rows sum to the same value (`V1+V2+V3 = V4+V5+V6 = V7+V8+V9`).

   Give Prolog facts and rules that define `semiMagic(L)`. `L` is the list of the integers 1 to 9, in some order. `semiMagic(L)` is true if the values in `L` represent a semi-magic square.

   (b) (10 points)

   A square is said to be "magic" if it is semi-magic and if the three columns of the square also sum to the same value (`V1+V4+V7 = V2+V5+V8 = V3+V6+V9`). Extend your solution to (a) to define a relation `magic(L)`. `magic(L)` is true if `semiMagic(L)` is true and if the three columns of the square `L` represents sum to the same value.