Homework 4

Due 29 October

Handout 8
CS242: Autumn 2003
22 October

Reading

1. Read Chapter 8, Control in Sequential Languages.

Problems

1. .... Exceptions

The following two versions of the `closest` function take an integer `x` and an integer tree `t` and return the integer leaf value from `t` that is closest in absolute value to `x`. The first is a straightforward recursive function, the second uses an exception.

```
datatype 'a tree = Leaf of 'a | Nd of ('a tree) * ('a tree);

fun closest(x, Leaf(y)) = y:int
|   closest(x, Nd(y,z)) = let val lf = closest(x,y) and rt = closest(x,z) in
                          if abs(x-lf) < abs(x-rt) then lf else rt end;

fun closest(x, t) =
   let
       exception Found
       fun cls (x, Leaf(y)) = if x=y then raise Found else y:int
       |   cls (x, Nd(y,z)) = let val lf = cls(x,y) and rt = cls(x,z) in
                              if abs(x-lf) < abs(x-rt) then lf else rt end
   in
       cls(x,t) handle Found => x
   end;
```

(a) Explain why both give the same answer.

(b) Explain why the second version may be more efficient.

2. .... Exceptions and Recursion

Here is an ML function that uses an exception called Odd.

```
fun f(0) = 1
|   f(1) = raise Odd
|   f(3) = f(3-2)
|   f(n) = (f(n-2) handle Odd => ~n)
```

The expression `~n` is ML for $-n$, the negative of the integer n.

When `f(11)` is executed, the following steps will be performed:

```
call f(11)
call f(9)
call f(7)
  ...
```

Write down the remaining steps that will be executed. Include only the following kinds of steps:

- function call (with argument)

- function return (with return value)
- raise an exception
- pop activation record of function off stack without returning control to the function
- handle an exception

Assume that if f calls g and g raises an exception that f does not handle, then the activation record of f is popped off the stack without returning control to the function f.

3. .... Tail Recursion and Exception Handling

Can we use tail recursion elimination to optimize the following program?

```
exception OddNum;
let fun f(0,count) = count
      | f(1,count) = raise OddNum
      | f(x,count) = f(x-2, count+1) handle OddNum => -1
```

Why or why not? Explain. This is a tricky situation – try to explain succinctly what the issues are and how they might be resolved.

4. .... Evaluation Order and Exceptions

Suppose we add an exception mechanism similar to the one used in ML to Pure Lisp. Pure Lisp has the property that if every evaluation order for expression e terminates, then e has the same value under every evaluation order. Does Pure Lisp with exceptions still have this property? (*Hint:* See if you can find an expression containing a function call $f(e_1, e_2)$ so that evaluating e_1 before e_2 gives you a different answer than evaluating the expression with e_2 before e_1 .)

5. .... Control Flow and Memory Management

An *exception* aborts part of a computation and transfers control to a handler that was established at some earlier point in the computation. A *memory leak* occurs when memory allocated by a program is no longer reachable, and the memory will not be deallocated. (The term "memory leak" is used only in connection with languages that are not garbage collected, such as C.) Explain why exceptions can lead to memory leaks, in a language that is not garbage collected.

6. .... Tail Recursion and Continuations

(a) Explain why a tail recursive function, as in

```
fun fact(n) =
    let fun f(n,a) = if n=0 then a
                     else f(n-1, a*n)
    in  f(n,1) end;
```

can be compiled so that the amount of space required to compute fact(n) is independent of n.

(b) The function f used in the following definition of factorial is "formally" tail recursive: the only recursive call to f is a call that need not return.

```
fun fact(n) =
    let fun f(n,g) = if n=0 then g(1)
                     else f(n-1, fn x=>g(x)*n)
    in  f(n, fn x => x) end;
```

How much space is required to compute fact(n), measured as a function of argument n? Explain how this space is allocated during recursive calls to f and when the space may be freed.

7. .... Continuations

In addition to continuations that represent the "normal" continued execution of a program, we can use continuations in place of exceptions. For example, consider the following function `f` that raises an exception when the argument `x` is too small.

```
exception Too_Small;
fun f(x) = if x<0 then raise Too_Small else x/2;
(1 + f(y)) handle Too_Small => 0;
```

If we use continuations, then `f` could be written as a function with two extra arguments, one for normal exit and the other for "exceptional exit," to be called if the argument is too small.

```
fun f(x, k_normal, k_exn) = if x<0 then k_exn() else k_normal(x/2);
f(y, (fn z => 1+z), (fn () => 0));
```

(a) Explain why the final expressions in each program fragment will have the same value, for any value of `y`.

(b) Why would tail call optimization be helpful when we use the second style of programming instead of exceptions?

8. .... Continuation-based Tree Comparison

This problem asks you to modify and run some continuation-passing Standard ML code. You should turn your answer in on paper along with the rest of your homework; this problem does not involve electronic submission of your code.

The algorithmic problem we will consider is the problem of comparing two trees to see if they have the same leaves in the same order. One way to do this comparison is to flatten the two tree structures and form a list of leaves for each tree, then compare the lists. However, this approach traverses both trees completely to flatten them, and then traverses both lists. Moreover, flattening each tree constructs a list with a cell for each leaf, consuming memory.

A more efficient solution (without resorting to "ugly" pointers and assignment) uses two recursive functions, one traversing each tree. We use `callcc` to capture the state of the recursive traversal of one tree, and `throw` to continue the other traversal, in a way that compares the leaves one at a time. This pattern is an example of the traditional control structure called coroutines, implemented using continuations.

Although the code is "elegant" in some sense, it is also very hard to understand if you are not used to working with continuations. Therefore, this problem asks you to add print statements to see how the code works and report based on your experiments. The code is shown at the end of this problem and is available on the web site, in the CS242 handouts directory.

To give a short summary, the main work is done by functions `searchA` and `searchB`, which operate as coroutines, each calling the other through the `resume` function. Function `searchA` traverses a tree, comparing each leaf with one produced by `searchB`. When called, `searchB` produces a leaf and returns a continuation so its traversal can be resumed when needed.

(a) Instrument `searchA` and `searchB` with print statements to see the order in which they are being called on this example:

```
val tree1 = node(leaf(1), node(leaf(2), leaf(3)));
val tree2 = node(node(leaf(1), leaf(2)), leaf(3));
compare(tree1,tree2);
```

Copy or print the output you get from your instrumented code for this example. Your output should include (1) which function is being executed, and (2) which leaf in which tree is being explored.

(b) In 2–4 sentences (no more), describe what `resumeA` and `resumeB` do. Say something more informative than what we've already told you in this problem statement.

(c) You have probably noticed that the code works well when you compare trees of the same size, but not as well when the leaves of one tree is a prefix of the other. For example, consider

```
val tree3 = node(tree1, node(tree2, tree1));
val tree4 = node(tree1, tree2);
```

Try experimenting with the code on such cases. Describe the output of the program in the situations that you find informative and describe a *simple* way of solving this problem. Your answer should be shorter than 10 sentences in total.

```
open SMLofNJ.Cont (* open module that provides continuations *)

datatype tree = leaf of int | node of tree*tree

datatype coA = A of (int* coB) cont  (* searchA wants an int and a B-continuation *)
and      coB = B of       coA  cont  (* searchB wants an A-continuation *)

fun resumeA(x, A k) = callcc(fn k' => throw k (x, B k'))
fun resumeB(   B k) = callcc(fn k' => throw k (A k'))

exception DISAGREE
exception DONE

fun searchA(leaf(x),(y, other: coB)) =
         if x=y then resumeB(other) else raise DISAGREE
|   searchA(node(t1,t2), other) = searchA(t2, searchA(t1, other))

fun searchB(leaf(x), other : coA) = resumeA(x,other)
|   searchB(node(t1,t2), other) = searchB(t2, searchB(t1, other))

fun startB(t: tree)  =  callcc(fn k => (searchB(t, A k); raise DONE))

fun compare(t1,t2) = searchA(t1, startB(t2))
```