Homework 5

Due 5 November

Handout 13
CS242: Autumn 2002
29 October

Reading

1. Read Chapter 7 (Scope, Function Calls, and Storage Management) and sections 1–3 of Chapter 8 (Control Mechanisms in Sequential Languages) of the Course Reader.

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?