Homework 4

Due 29 October

Handout 10
CS242: Autumn 2002
22 October

Reading

1. Read Chapter 7, Scope, Functions, and Storage Management.

Problems

1. .... Activation Records for Inline Blocks

You are helping a friend debug a C program. The debugger (e.g., `gdb`) lets you set breakpoints in the program, so the program stops at certain points. When the program stops at a breakpoint, you can examine the values of variables. If you want, you can compile the programs given in this problem and run them under a debugger yourself. However, you should be able to figure out the answers to the questions by thinking about how activation records work.

(a) Your friend comes to you with the following function (`times`), which is supposed to calculate and print the product of its inputs, `x` and `n` (i.e. `x * n`). Your friend then writes a small test program to exercise the `times` function:

```
1: void times(int x, int n)
2: {
3:   int i;
4:   int prod = 0;
5:   for (i = 0; i < n; i++)
6:   {
7:     int prod = prod + x;
8:   }
9:   printf ("The product is %d.\n", prod);
10:}
11:int main(void)
12:{
13:  times(2, 3);
14:  return 0;
15:}
```

Your friend complains that this program just doesn't work and shows you some sample output:

```
cardinal:~> gcc -g test.c
cardinal:~> ./a.out
The product is 0.
```

Being a careful student, your friend has also used the debugger to try to track down the problem. Your friend sets breakpoints at lines 7 and 9 and looks at the address of `prod`. Here is the debugger output:

```
cardinal:~> gdb a.out
(gdb) br test.c:7
(gdb) br test.c:9
(gdb) r
Starting program:

Breakpoint 1, times (x=2, n=3) at test.c:7
```

```
7             int prod = prod + x;
(gdb) print &prod
$1 = (int *) 0xbffff59c
(gdb) del 1
(gdb) c
Continuing.

Breakpoint 2, times (x=2, n=3) at test.c:9
9         printf ("The product is %d.\n", prod);
(gdb) print &prod
$2 = (int *) 0xbffff5a0
```

Your friend swears that the computer must be broken, since it is changing the address of the variable `prod`. Using the information provided by the debugger and the concept of activation records, give your friend a 3 or 4 sentence explanation of what the problem is and why the output is 0.

(b) Your explanation must not have been that good, because your friend attempts to fix the program and produces the following:

```
void times(int x, int n)
{
  int i;
  for (i = 0; i < n; i++)
  {
    int prod = prod + x;
  }
  {
    int prod;
    printf ("The product is %d.\n", prod);
  }
}
int main(void)
{
  times(2, 3);
  return 0;
}
```

This program sill does not work. Explain why it does not work (in 3 sentences or less), and write a formula for the printed value of `prod`. Your formula can include constants with an unknown value as long as you explain where the value of those constants comes from (e.g., σ, where σ is an uninitialized value on the stack).

(c) Your friend makes one last stab at getting the program to work, and produces the following;

```
void times(int x, int n)
{
  {
    int prod = x * n;
  }

  {
    int prod;
    printf ("The product is %d.\n", prod);
  }
}
int main(void)
{
  times(2, 3);
  return 0;
}
```

This program seems to work. Why?

(d) Imagine that line 6 of the program in part (c) were split into three lines:

```
6a:           {
6b:           ...
6c:           }
```

Write a single line of code to replace the ... that would guarantee this program would NEVER print the right product.

2. .... Time and Space Requirements

This question asks you to compare two functions for finding the middle element of a list. (In the case of an even-length list of $2n$ elements, both functions return the $n+1$st.) The first uses two local recursive functions, `len` and `get`. The `len` function finds the length of a list and `get(n,l)` returns the `n`th element of list `l`. The second middle function uses a subsidiary function `m` that recursively traverses two lists, taking two elements off the first list and one off the second until the first list is empty. When this occurs, the first element of the second list is returned.

```
exception Empty;

fun middle1(l) =
   let fun len(nil) = 0
       |   len(x::l) = 1+len(l)
       and get(n,nil) = raise Empty
       |   get(n,x::l) = if n=1 then x else get(n-1,l)
   in
       get((len(l) div 2)+1, l)
   end;

fun middle2(l) =
   let fun m(x,nil) = raise Empty
       |   m(nil,x::l) = x
       |   m([y],x::l) = x
       |   m(y::(z::l1),x::l2) = m(l1,l2)
   in
       m(l,l)
   end;
```

Assume that both are compiled and executed using a compiler that optimizes use of activation records or "stack frames."

(a) Describe the approximate running time and space requirements of `middle1` for a list of length n. Just count the number of calls to each function and the maximum number of activation records that *must* be placed on the stack at any time during the computation.

(b) Describe the approximate running time and space requirements of `middle2` for a list of length n. Just count the number of calls to `m` and the maximum number of activation records that *must* be placed on the stack at any time during the computation.

(c) Would an iterative algorithm with two pointers, one moving down the list twice as fast as the other, be significantly more or less efficient than `middle2`? Explain briefly in one or two sentences.

3. .... Parameter passing comparison

For the following Algol-like program, write the number printed by running the program under each of the listed parameter passing mechanisms. In pass-by-value/result, the formal parameter is initialized to the value of the actual. Changes to the formal parameter within the subroutine do not immediately change the value of the actual parameter. When the subroutine returns, the value of the formal parameter is then copied back into the actual. This parameter-passing scheme is sometimes referred to as copy-in/copy-out.

```
begin
   integer i;

   procedure pass ( x, y );
      integer x, y;  // types of the formal parameters
      begin
         x := x + 1;
         y := x + 1;
         x := y;
         i := i + 1
      end

   i := 1;
   pass (i, i);
   print i
end
```

(a) pass-by-value

(b) pass-by-reference

(c) pass-by-value/result

4. .... Static and Dynamic Scope

Consider the following program fragment, written both in ML and in pseudo-C:

```
1  let x = 2 in                      { int x = 2; {
2    let val fun f(y) = x + y in       int f (int y) { return x + y; } {
3      let val x = 7 in                  int x = 7; {
4        x +                               x +
5            f(x)                              f(x);
6      end                               }
7    end                               }
8  end;                              }
```

The C version would be legal in a version of C with nested functions.

(a) Under static scoping, what is the value of $x + f(x)$ in this code? During the execution of this code, the value of x is needed three different times (on lines 2, 4, and 5). For each line where x is used, state what numeric value is used when the value of x is requested and explain why these are the appropriate values under static scoping.

(b) Under dynamic scoping, what is the value of $x + f(x)$ in this code? For each line where x is used, state which value is used for x and explain why these are the appropriate values under dynamic scoping.

5. .... Eval and Scope

Many compilers look at programs and eliminate any unused variables. For example, in the following program, x is unused so it could be eliminated:

```
let x = 5 in f(0) end
```

Some languages, including Lisp and Scheme, have a way to construct and evaluate expressions at run-time. Constructing programs at run-time is useful in certain kinds of problems, such as symbolic mathematics and genetic algorithms.

The following program evaluates the string bound to s, inside the scope of two declarations:

```
let s = read_text_from_user() in
  let x = 5 and y = 3 in eval s end
end
```

If s were bound to `"1+x*y"` then `eval` would return 16. Assume that `eval` is a special language feature and not simply a library function.

(a) The "unused variable" optimization and the "eval" construct are not compatible. The identifiers `x` and `y` do not appear in the body of the inner `let` (the part between `in` and `end`), yet an optimizing compiler cannot eliminate them because the `eval` *might* need them. In addition to the values 5 and 3, what information does the language implementation need to store for `eval` that would not be needed in a language without `eval`?

(b) A clever compiler might look for `eval` in the scope of the `let`. If `eval` does *not* appear, then it may be safe to perform the optimization. The compiler could eliminate any variables that do not appear in the scope of the `let` declaration. Does this optimization work in a statically scoped language? Why or why not?

(c) Does the optimization suggested in part (b) work in a dynamically scoped language? Why or why not?

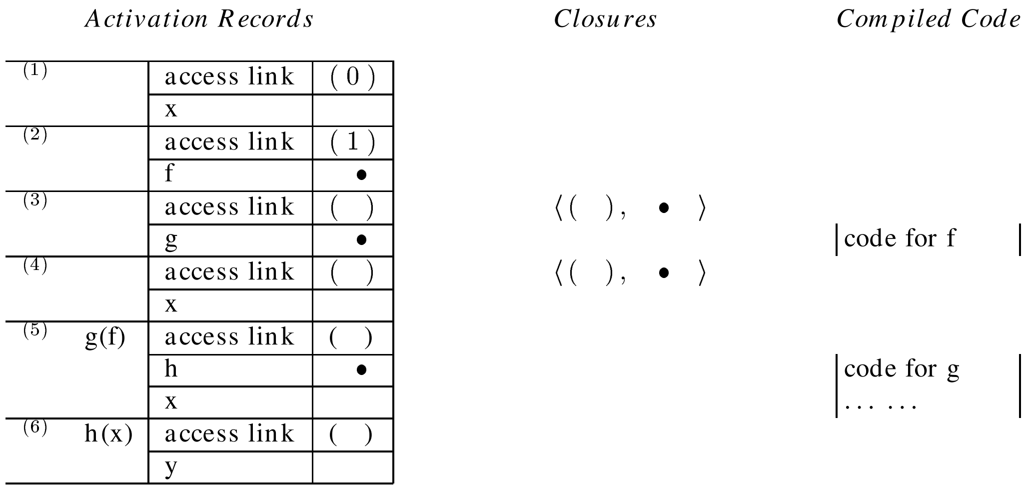
6. .... Function Calls and Memory Management

This question asks about memory management in the evaluation of the following statically-scoped ML expression.

```
val x = 5;
fun f(y) = (x+y)-2;
fun g(h) = let val x = 7 in h(x) end;
let val x = 10 in  g(f) end;
```

(a) Fill in the missing information in the following depiction of the run-time stack after the call to h inside the body of g. Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.



(b) What is the value of this expression? Why?

7. .... Closures and Returning a Function

Consider the following code:

```
fun g(f) =
    let val x : int ref = ref (f(1))
    in
        fn(y) => (x := (!x) * y; !x)
    end;
val x = 1;
fun s(y) = y + x;
val h = g(s);
val z = h(3);
```

(a) What are the types of g, h, and z?

(b) What is the value of z?

(c) Draw the run-time structures that result from the execution of this code, by adding to the following diagram.