```
(*** CSI 3120 Assignment 5 ***)
(*** There is no OCaml programming on this assignment, so you don't have
        to fill in any answers in this file.  You can hand in, for example,
        a Word file or a .pdf file containing a scan of handwritten
        solutions. ***)
(*** Be sure to include YOUR NAME and YOUR STUDENT ID ***)

(* QUESTION 1. Dynamic Scope *)
(* Consider the following program from LAB 5 again.
   This exercise is similar, but with different call sequences.

{ begin
  int x = ...
  int y = ...
  int z = ...
  int f ()
        { int a = ...
        int y = ...
        int z = ...
        ...code for f...
        }
  int g ()
        { int a = ...
        int b = ...
        int z = ...
        ...code for g...
        }
  int h ()
        { int a = ...
        int x = ...
        int w = ...
        ...code for h...
        }
  ...code for main (outer) block...
}

   Assume dynamic scope again.  Given the following calling
   sequences, what variables are visible during execution of the
   last function call in each sequence?  Include with each
   visible variable the name of the block where it is declared
   (main, f, g, or h).  Draw the full activation stack.  In each
   activation record, include the local variables and the
   control link only.
   (a) main calls h; h calls f.
   (b) main calls f; f calls h; h calls g.
   (c) main calls h; h calls g; g calls f.
 *)

(i)
------
x |  |
------
y |  | (main)
------
z |  |
------

------
a |  |
------
x |  | (h)
------
w |  |
------

------
a |  |
------
y |  | (f)
------
z |  |
------

Visible variables:
a in block f
w in block h
x in block h
y in block f
z in block f

(ii)
------
x |  |
------
y |  | (main)
```

```
------
z |  |
------

------
a |  |
------
y |  | (f)
------
z |  |
------

------
a |  |
------
x |  | (h)
------
w |  |
------

------
a |  |
------
b |  | (g)
------
z |  |
------
```

Visible variables:
a visible in g
b visible in g
x visible in h
w visible in h
y visible in f
z visible in g

```
(* QUESTION 2. Parameter Passing *)
(* Consider the following pseudo-Algol code.

  begin
        procedure pass(x, y);
        begin
```

```
            x := x + 1;
            y := x + 1;
            x := y;
            end

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

Write the number printed by running the program under each of the listed paramter passing mechanisms. Pass-by-value-result is explained in Question 3 of Lab 5.

(a) pass-by-value
        Prints 1
(b) pass-by-reference
        Prints 1
(c) pass-by-value-result
        Prints 2
*)

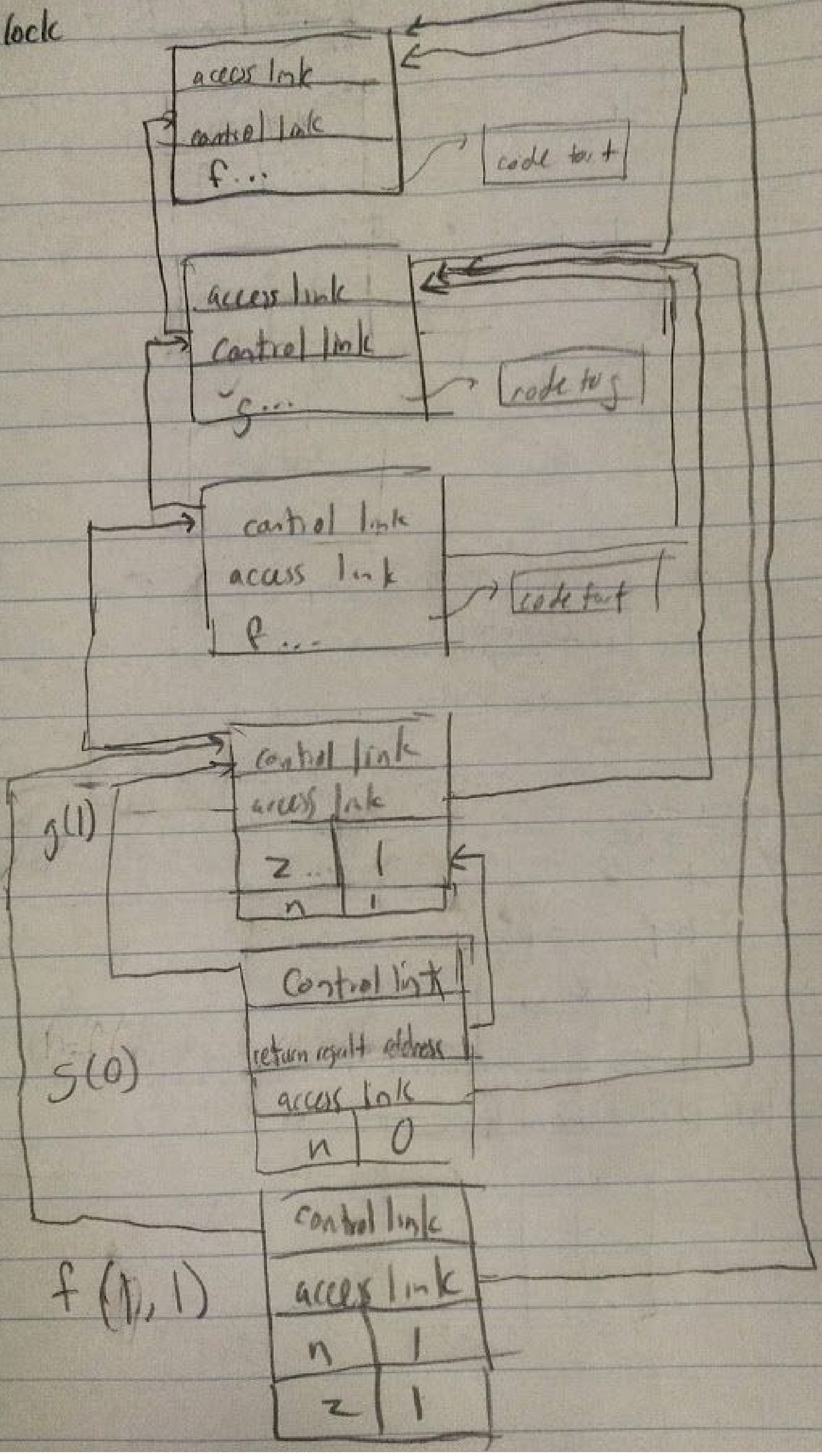
(* QUESTION 3. Tail Recursion *)
(* Consider the tail recursive OCaml program below. *)

```
let mult_tr (a:int) (b:int) =
  let rec mult' (a:int) (b:int) (result:int) =
            if a = 0 then 0
            else if a = 1 then b + result
            else mult' (a-1) b (result+b)
  in
  mult' a b 0

fun mult (a,b) = {
            result := 0;
            while not(a=0) do
                        a := a-1;
                        result := result + b;
            return result;
};
```

(* Translate this program to an equivalent one that uses a while loop

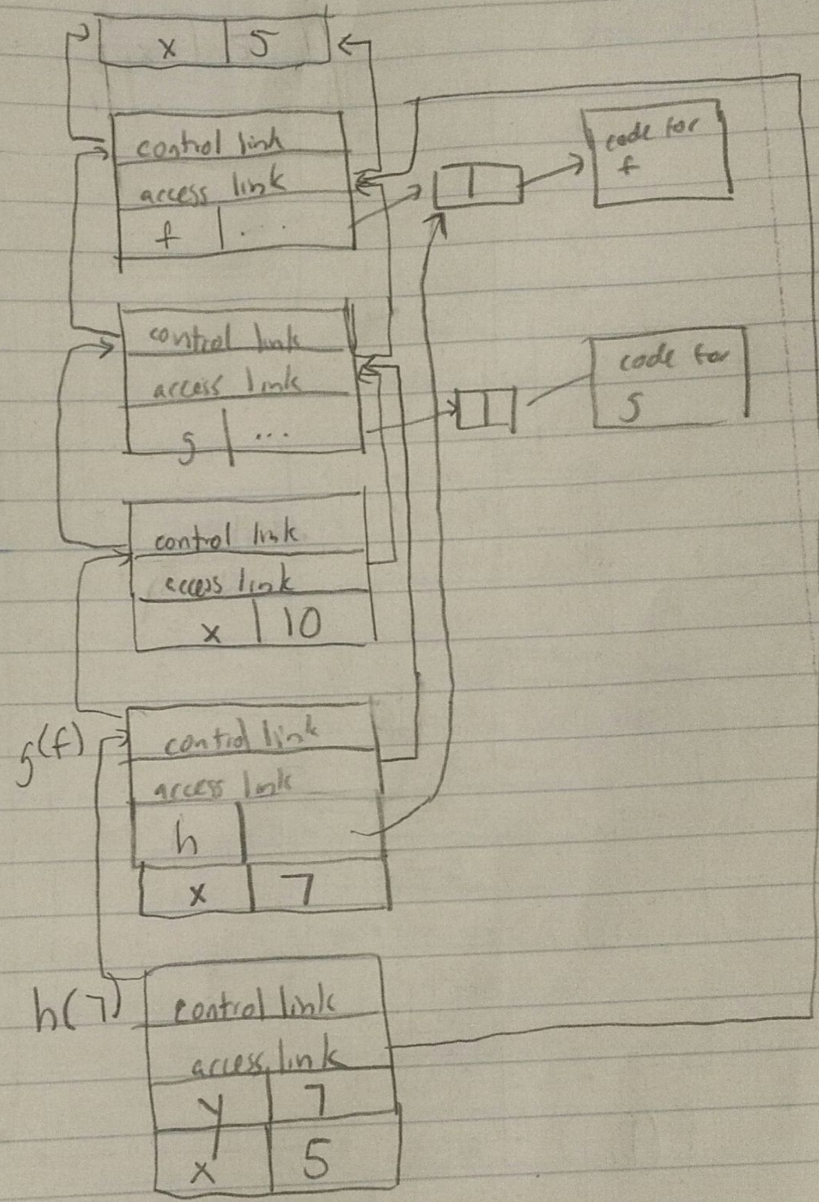
```
   instead of recursion.  (See page 34 of the course notes for Chapter
   7 of the Mitchell textbook.) Use the programming language that was
   defined in the course notes on Axiomatic Semantics (assignment
   statements, if statements, while loops, and sequences of statements
   separated by a semi-colon).  Let P be the name of your program.
   The following Hoare triple should be true about your program.

   { a >= 0 } P { n = a * b }.

   (You don't have to prove it.  Just make sure that your program is
   correct, and terminates whenever the precondition is satisfied.)
 *)

(* QUESTION 4. Activation Records and Recursive Function Calls *)

(* Consider the OCaml code below.  Ignore the first line and draw the
   activation stack for the execution of this code starting with the
   declaration of the first function f and ending with the activation
   record that is created when execution arrives at the function call
   (f 1 1).

   In your activation records, include the access links, parameters,
   and local variables.

   If any activation records need to be popped off the stack, do not
   erase them, but instead mark them as popped and continue below
   them.  *)

let _ =
  let f (x:int) (y:int) = x*y in
  let rec g (n:int) =
          if n = 0 then 1
          else (let z = g (n-1) in f n z) in
  let f (x:int) (y:int) = x+y in
  g 1
```

4. outer block



```
(* QUESTION 5. Activation Records and Functions as Arguments *)

(* Consider the OCaml code below.  Ignore the first line and draw the
   activation stack for the execution of this code starting with the
   first declaration of the variable x and ending with the activation
   record that is created when the function h is called inside the
   function 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 the compiled code.

   Again, in your activation records, include the access links,
   parameters, and local variables. *)

let _ =
  let x = 5 in
  let f (y:int) = (x+y)-2 in
  let g (h:int->int) =
        let x = 7 in h x in
 let x = 10 in
  g f
```

5. outer block


x | 5
control link
access link
f | ...
code for f
control link
access link
g | ...
code for g
control link
access link
x | 10
g(f)
control link
access link
h
x | 7
h(7)
control link
access link
y | 7
x | 5