Homework Assignment #2: Control Structures, Subprograms, Scoping

(due on June 2nd at the beginning of class)

Problem #1: Iterative Control Structures (60 points)

Consider the following **for** loop in ALGOL'60-like syntax:

```
for i := 1 step 2 until 13 do
  begin
   print i
  end
```

The code above prints out values: **1, 3, 5, 7, 9, 11, 13**. Based on this fact you should be able to determine the whole semantic of the ALGOL'60 **for** statement. (Assume that `until` corresponds to a "lower or equal than".)

The semantics of the loop can be inferred as being:

```
      i = initial_expression
loop: if i > until_expression goto out
      loop_body
      i = i + step_expression
      goto loop
out:  noop
```

Now consider the following fragment of code:

```
j = 1
for i := j + 1 step i+j until 4*j do
  begin
   print i
   j := j + 1
  end
```

a. *(20 points)* Like we did in class for C, write a operational semantic description of the above loop based on the assumptions that both the **step** and the **until** expressions are evaluated only once (at loop entry, after the initialization of the loop variable `i` has occurred). What does the code output?

Hint: think of the `step` and `until` expressions as stored in temporary variables.

```
      j = 1
      i = j + 1
      tmp_step = i + j
      tmp_until = 4 * j
loop: if i > tmp_until goto out
      print i
      j = j + 1
      i = i + tmp_step
      goto loop
out:  noop
```

The program prints out: **2**

b. *(20 points)* Write an operational semantic description of the code assuming that both the **step** and the **until** expressions are evaluated at the beginning of each iteration (before the body of the loop is executed and before the loop variable is incremented for that iteration). What does the code output?

```
      j = 1
      i = j + 1
loop: tmp_until = 4 * j
      tmp_step = i+j
      if i > tmp_until goto out
      print i
      j = j + 1
      i = i + tmp_step
      goto loop
out:  noop
```

The program prints out: **2, 5, 12**

c. *(20 points)* Write an operational semantic description of the code assuming that the **step** expression is evaluated only once a loop entry after initialization of the loop variable, and the **until** expression is evaluated at the beginning of each iteration. What does the code output?

```
      j = 1
      i = j + 1
      tmp_step = i+j
loop: tmp_until = 4*j
      if i > tmp_step goto out
      print i
      j = j + 1
      i = i + tmp_step
      goto loop
out:  noop
```

The program prints out: **2, 5, 8, 11, 14, ...** (doesn't terminate)

Problem #2: Control Structures (30 points)

Consider the following C program segment. Rewrite it without using `goto`, `break`, or `continue`.

```
j = -3;
for (i=0; i < 3; i++) {
  switch (j+2) {
    case 3:
    case 2: j--; break;
    case 0: j+=2; break;
    default: j=0;
  }
  if (j > 0) break;
  j = 3-i;
}
```

```
j = -3;
for (i=0; i < 3; i++) {
    if ((j+2 == 3) || (j+2 == 2))
```

```
            j--;
        else {
          if (j+2 == 0)
              j+=2;
          else
              j=0;
        }
        if (j <= 0)
            j = 3-i;
        else
            i = 3;
    }
```

Problem #3: Parameter passing (40 points)

Consider the following program written in C-like syntax:

```
void swap(int a, int b) {
    int temp;

    temp = a;
    a = b;
    b = temp;
}

int main() {
    int value = 4, list[5] = {4,3,2,1,0};

    swap(value,list[0]);
    printf("%d and [%d,%d,%d,%d,%d]\n",
                value,list[0],list[1],list[2],list[3],list[4]);

    swap(list[0],list[1]);
    printf("%d and [%d,%d,%d,%d,%d]\n",
                value,list[0],list[1],list[2],list[3],list[4]);

    swap(value,list[value]);
    printf("%d and [%d,%d,%d,%d,%d]\n",
                value,list[0],list[1],list[2],list[3],list[4]);

return 0;
}
```

For each of the following parameter-passing methods, what is the output of the program:

a. *(10 pts)* Passed by value.

```
4 and [4,3,2,1,0]
4 and [4,3,2,1,0]
4 and [4,3,2,1,0]
```

b. *(10 pts)* Passed by reference.

```
4 and [4,3,2,1,0]
4 and [3,4,2,1,0]
0 and [3,4,2,1,4]
```

c. *(10 pts)* Passed by name.

4 [4,3,2,1,0]
4 [3,4,2,1,0]
0 [4,4,2,1,0]

d. *(10 pts)* Passed by value-result.

4 and [4,3,2,1,0]
4 and [3,4,2,1,0]
0 and [3,4,2,1,4]

Problem #4: Scoping (40 points)

Consider the following program written in C-like syntax:

```
int x = 0;

void inc_x(int n) {
    x += n;
}

void print_x() {
   printf("%d\n",x);
}

void first() {
   inc_x(3);
   print_x();
}

void second() {
   int x=10;
   inc_x(2);
   print_x();
}

int main() {
   inc_x(1);
   first();
   print_x();
   second();
   print_x();
}
```

a. *(20 points)* What does this program print if the language uses static scoping?

4
4
6
6

b. *(20 points)* What does this program print if the language uses dynamic scoping?

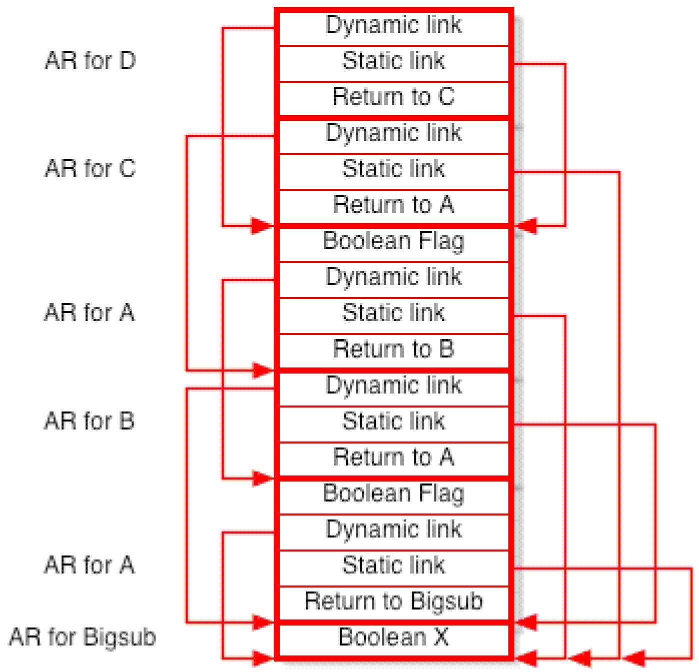
4
4
12
4

Problem #5: Activation Records (20 points)

Consider the following fragment of Ada-like code:

```
procedure Bigsub is
   procedure A(Flag : Boolean) is
      procedure B is
         begin
         ...
         A(false);
         end; -- of B
      begin -- of A
      if flag
         then B;
         else C;
      ...
      end; -- of A
   procedure C is
      procedure D is
         begin
         ...       <------------ HERE
         end; -- of D
      begin -- of C
      ...
      D;
      end; -- of C
   begin -- of Bigsub
   X : Boolean := true;
   ...
   A(X);
   ...
   end; -- of Bigsub
```

Draw the stack with all activation record instances when execution reaches the point of the program marked by the red arrow, including static and dynamic links as well as procedure parameters and local variables, like we did in class. Assume that procedure `Bigsub` is at level 1 (i.e., the first activation record at the bottom of the stack).


AR for D
Dynamic link
Static link
Return to C
AR for C
Dynamic link
Static link
Return to A
Boolean Flag
AR for A
Dynamic link
Static link
Return to B
AR for B
Dynamic link
Static link
Return to A
Boolean Flag
AR for A
Dynamic link
Static link
Return to Bigsub
AR for Bigsub
Boolean X