For these questions, use a three-address intermediate representation like the one we used in class. The instructions are:

```
x = y
x = n
x = &y          (get address of variable y)
x = a op b,     where a and b are names or constants, and op is any arithmetic,
                comparison, or boolean operator
JUMP label
CJUMP x, Lt, Lf      (jump to Lt if x is true, Lf  otherwise)
x = LOADIND y       (get value in location contained in y)
ERROR               (return an error)
```

You may use any of the following translation schemes, which were discussed in class, except if a question requires you to use a particular one:

[S] = instruction list to compute statement S
[e] = pair containing instruction list to compute expression e and variable name giving location of result
$\mathbf{[S]_{blab,clab}}$ = instruction list for S, in context of while or switch statement, where blab is the break label (label after the enclosing while or switch) and clab is the continue label (label to start next iteration of while).
$\mathbf{[e]_x}$ = instruction list to calculate value of e and place it in x
$\mathbf{[e]_{tlab,flab}}$ = instructions to calculate boolean-valued expression e and jump to tlab if it is true, flab otherwise. (This is called the "short-circuit evaluation" scheme.)

1. Generate code for the repeat-until statement: "repeat S until e" executes S and tests e, and repeats until e becomes true. Thus, it is equivalent to "S; while !e do S". Do this in two ways: (a) Using the regular scheme [e] to evaluate the condition; and (b) Using the short-circuit evaluation scheme for the condition.

(a)

```
let (I, t) = [e]
    L1, L2 = new labels
in
   L1:  [S]
        I
        CJUMP t, L1, L2
   L2:
```

(b)

```
let (I, t) = [e]
    L1, L2 = new labels
in
   L1:  [S]
        [e]L2,L1
   L2:
```

2. Generate code for a multiple assignment statement: (x1, x2) = (e1, e2), which does both assignments "in parallel." Note that this is not that same as doing one assignment followed by the other, because either variable x1 or x2 may appear in expressions e1 and e2. Use evaluation scheme $[e]_x$ where appropriate.

```
let t = new variable
in      [e1]t
        [e2]x2
        x1 = t
```

3. Give two schemes for conditional expression e1 ? e2 : e3, which gives the value of e2 if e1 is true, or e3 if e1 is false. The two schemes you should provide are (a) the standard [e1 ? e2 : e3], and (b) the assignment scheme $[e1 ? e2 : e3]_x$. You may use the standard scheme for the condition; extra credit will be awarded for using short-circuit evaluation for the condition.

(Note: both solutions use the short-circuit scheme.)

```
[e1 ? e2 : e3] =
  let (Ii, ti) = [ei], i=2,3
      t = new variable
      L1, L2, L3 = new labels
  in  (       [e1]L1,L2
      L1:     I2
              t = t2
              JUMP L3
      L2:     I3
              t = t3
      L3:             ,   t)
```

```
[e1 ? e2 : e3]x =
  let L1, L2, L3 = new labels
  in          [e1]L1,L2
        L1:   [e2]x
              JUMP L3
        L2:   [e3]x
        L3:
```

4. Some languages have array declarations of the form: int A[lo:hi], to declare an array with indices lo, lo+1, …, hi. (A C-type declaration int A[10] would become int A[0:9].) Suppose we know (from the symbol table) that A has been declared as int A[lo:hi]. Give a translation scheme for A[i] as a right-hand value (i.e. get the contents of A[i]). The compiled code should halt with an error (i.e. execute the ERROR instruction) if i< lo or i>hi. As in class, you will need to use the x = &A and LOADIND instructions.

```
[ A[e] ] =  let (I,t) = [e]
                t1, ..., t5 = new variables
                L1, ..., L4 = new labels
            in
                    t1 = &A
                    I
                    t2 = t < lo
                    CJUMP t2, L2, L1
```

```
L1:     t2 = t > hi
        CJUMP t2, L2, L1
L3:     t2 = t-lo
        t3 = t2 * size of A's elements
        t4 = t1+t3
        t5 = LOADIND t4
        JUMP L4
L2:     ERROR
L4:
```