# EECS 665
# Final Exam Review Questions

1. Give the three-address code (like the quadruples in the csem assignment) that could be emitted to translate the following assignment statement. However, you may **not** use the [ ] quadruple operation. Assume that `a` is a global integer array with 5 rows and 10 columns. Assume that `v` is a local integer variable (the first one defined) and `i` and `j` are integer parameters (the first two) passed to the function. Further assume that the arrays are stored in *row-major* order.

```
v = a[i][j];
```

**Solution:**

$$v = @(base_a + (10 * i + j) * 4)$$

(a) Code Generated by *csem* in Project-2.

```
formal 4
formal 4
localloc 4
t1 = local 0
t2 = global a
t3 = 10
t4 = param 0
t5 = @t4
t6 = t3 *i t5
t7 = param 1
t8 = @i t7
t9 = t6 +i t8
t10 = 4
t11 = t9 *i t10
t12 = t2 +i t11
t13 = @t12
t14 = t1 =i t13
```

(b) OR More independently from our project-2, this is also the 3-address code :

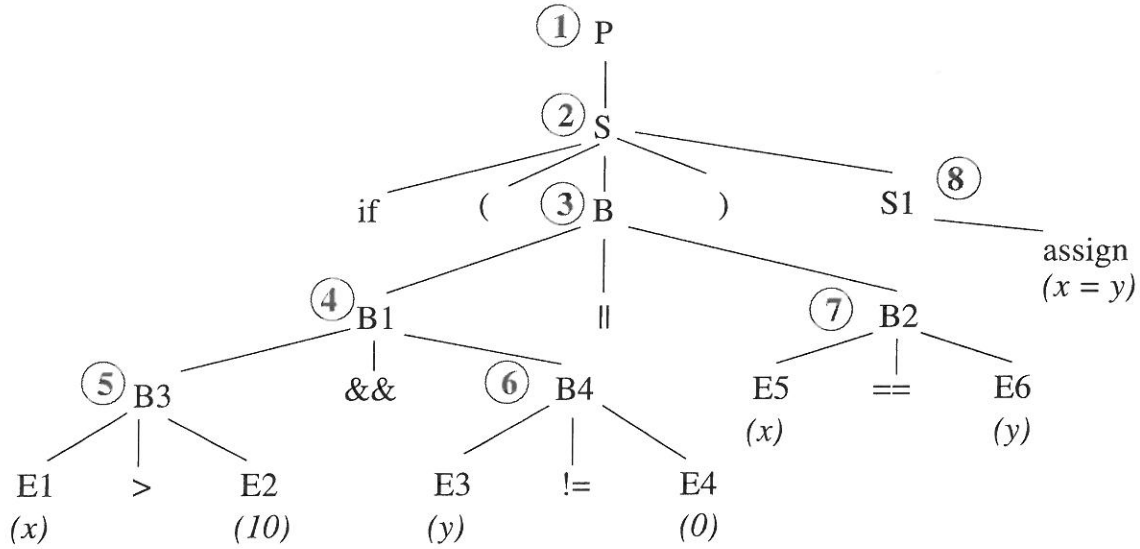
```
t1 = param i;
t2 = @t1;
t3 = 10 * t2;
t4 = param j;
t5 = @t4;
t6 = t3 + t5;
t7 = t6 * 4;
t8 = global a;
t9 = t8 + t7;
t10 = @t9;
t11 = local v;
t12 = t11 = t10;
```

2. Use Figures 6.36 and 6.37 to translate the following statement in two passes (no backpatching).

$$if(x > 10 \quad \&\& \quad y! = 0 \quad || \quad x == y) \quad x = y;$$

**Answer:**

A. Use a top-down pass to evaluate all the non-synthesized attributes.




1. S.*next* = L1

2. B.*true* = L2
   B.*false* = $S_1$.*next* = L1

3. $B_1$.*true* = L2
   $B_1$.*false* = L3
   $B_2$.*true* = L2
   $B_2$.*false* = L1

4. $B_3$.*true* = L4
   $B_3$.*false* = L3
   $B_4$.*true* = L2
   $B_4$.*false* = L3

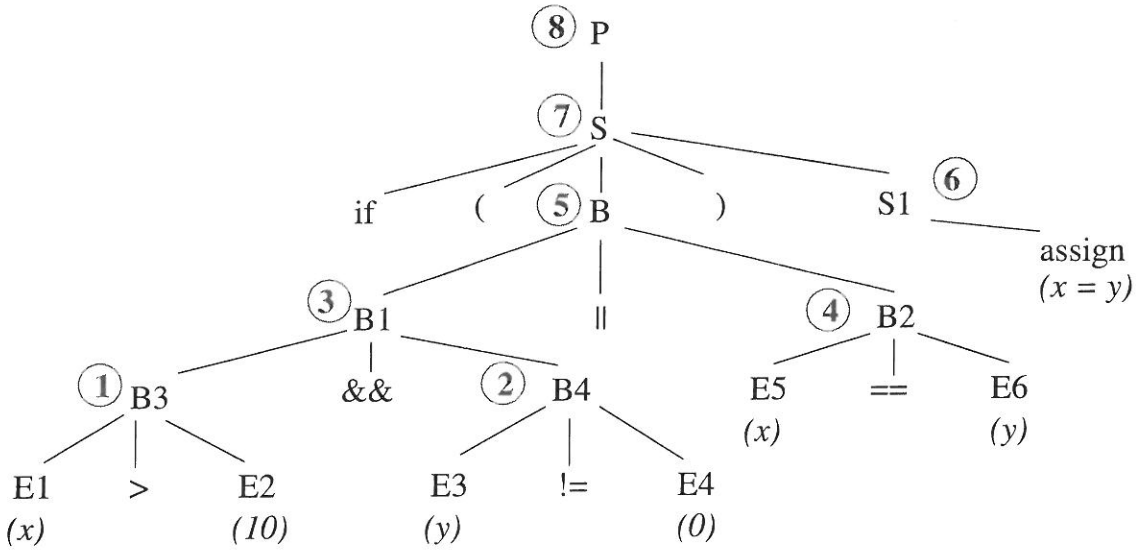
5. —

6. —

7. —

8. —

B. Use a bottom-up pass to evaluate the synthesized attributes




1. $B_3.code =$
   if x>10 goto L4
   goto L3

2. $B_4,code =$
   if y! =0 goto L2
   goto L3

3. $B_1.code =$
   if x>10 goto L4
   goto L3
   L4:
   if y! =0 goto L2
   goto L3

4. $B_2.code =$
   if x==y goto L2
   goto L1

5. $B.code =$
   $B_1.code$ || L3: || B2.$code$

6. $S_1.code =$
   x = y

7. S.$code =$
   B.$code$ || L2: || $S_1.code$

8. P.$code =$
   S.$code$ || L1:

   which, when expanded looks like:

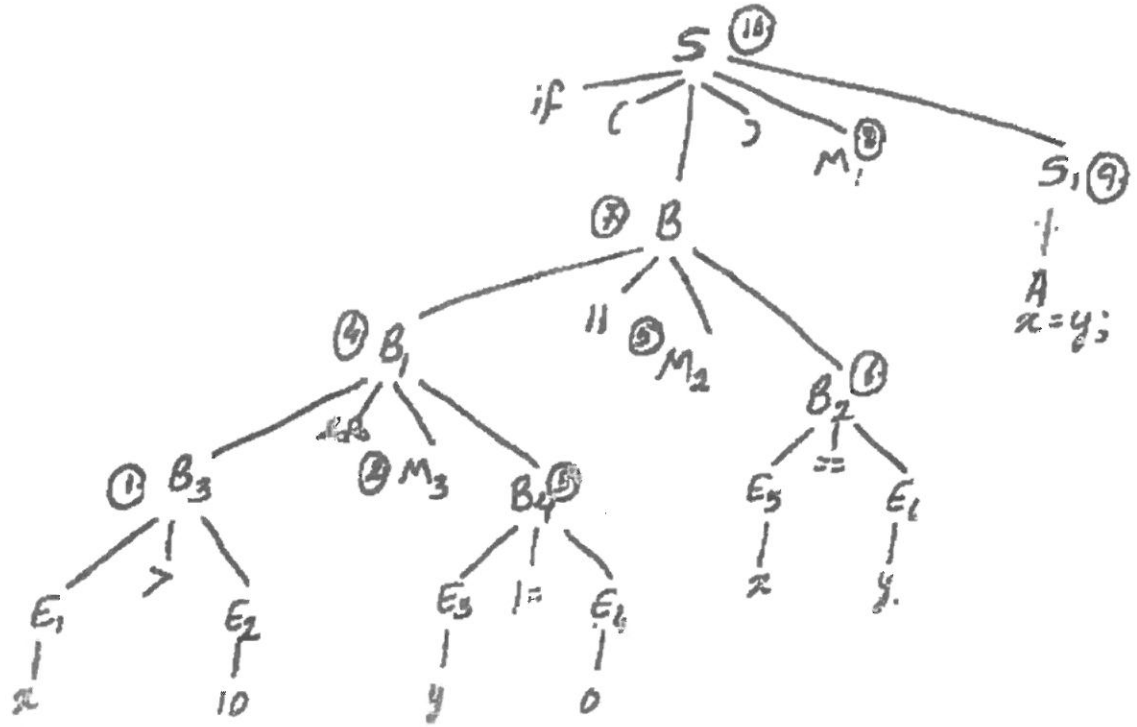
   $B_1.code$ ||
   L3: $B_2.code$ ||
   L2: $S_1.code$ ||
   L1:

   which, when further expanded, looks like:

   if x>10 goto L4
   goto L3
   L4: if y! =0 goto L2
   goto L3
   L3: if x==y goto L2
   goto L1
   L2: x = y
   L1:

3. Use Figures 6.43 and 6.46 to translate the expression in a single pass (use backpatching). Start outputing the code from address 20.

$$if(x > 10 \quad \&\& \quad y! = 0 \quad || \quad x == y) \quad x = y;$$




① $B_3.true = \{20\}$
$B_3.false = \{21\}$

② $M_3.inst = 22$

③ $B_4.true = \{22\}$
$B_4.false = \{23\}$

④ $\{20\} \longrightarrow 22$
$B_1.true = \{22\}$
$B_1.false = \{21, 23\}$

$M_2.inst = 24$

| | |
|---|---|
| 20: | if $x > 10$ goto 22 |
| 21: | goto 24 |
| 22: | if $y! = 0$ goto 26 |
| 23: | goto 24 |
| 24: | if $x == y$ goto 26 |
| 25 | goto .... |
| 26 | $x = y$ |

3.

(6) $B_2$ true = {24}
$B_2$.false = {25}

(7) {21, 23} → 24
B.true = {22, 24}
B.false = {25}

(8) M.inst = 26

(9) print x = y

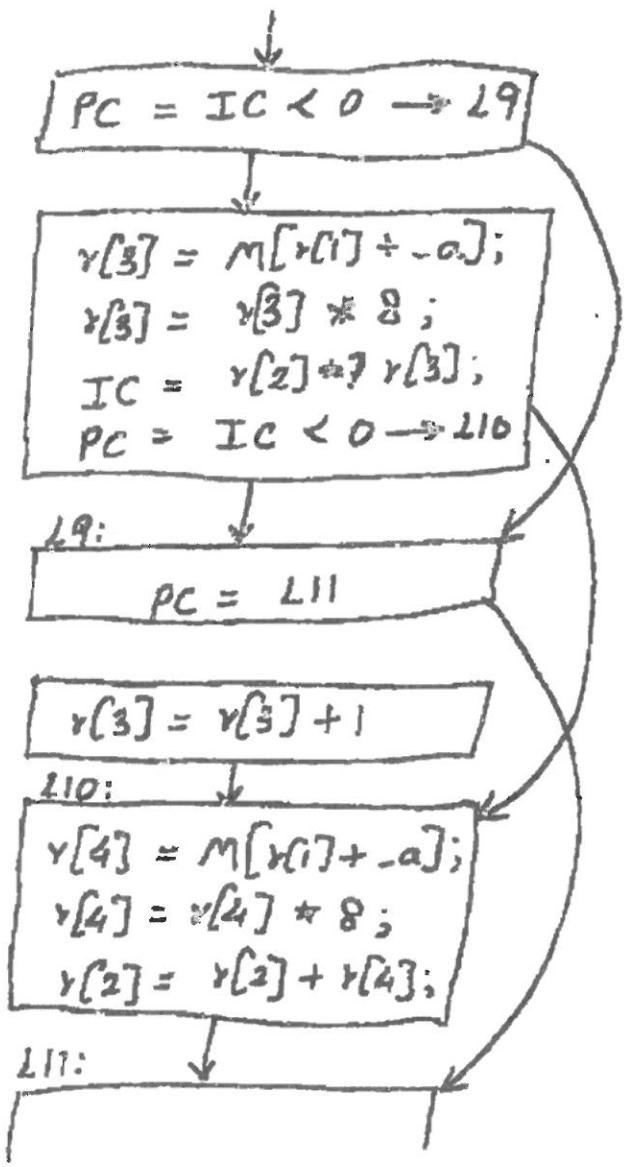
(10) {22, 24} → 26
S.next = {25}.

4. Below is a syntax directed definition, where **A**, **B**, **C**, and **D** are nonterminals and **a**, **b**, and **c** are terminals. Indicate for each rule if the attribute being assigned is synthesized or inherited attribute.

| Production | Semantic Rules | Synthesized or Inherited? |
|---|---|---|
| A $\rightarrow$ B C | C.a := B.a | I |
| | A.a := C.b | S |
| B $\rightarrow$ D a | D.a := B.a | I |
| C $\rightarrow$ $C_1$D | $C_1$.a := D.a | I |
| | C.b := $C_1$.b | S |
| C $\rightarrow$ b | C.b := b.val | S |
| D $\rightarrow$ c | D.a := c.val | S |

5. The following RTLs represent instructions from part of a function that a compiler has generated for a machine with no delay slots. Identify the leaders and depict the RTLs in a flow graph of basic blocks (note that each RTL should appear in a basic block).

```
        PC = IC < 0 → L9;          /* this is the only branch to L9 */
        r[3] = M[r[1] + _a];
        r[3] = r[3] * 8;
        IC = r[2] ? r[3];
        PC = IC < 0 → L10;         /* this is the only branch to L10 */
L9
        PC = L11;
        r[3] = r[3] + 1;
L10
        r[4] = M[r[1] + _a];
        r[4] = r[4] * 8;
        r[2] = r[2] + r[4];        r[4]:
L11
```



6. The following RTLs represent **SPARC** instructions. The **SPARC**, like most RISC machines, is a 3 address machine and can only access memory by using load or store instructions. Apply the instruction selection optimization (as described in class) to the following RTLs. Show each step of combining RTLs separately. (10 points)

| | | | |
|---|---|---|---|
| 1 | | r[1] = r[30] + j; | |
| 2 | {1} | r[2] = M[r[1]]; | r[1]: |
| 3 | | r[3] = 4; | |
| 4 | | r[4] = 1; | |
| 5 | {2,4} | r[2] = r[2] + r[4]; | r[4]: |
| 6 | {3} | r[3] = r[14] + r[3]; | r[3]: |
| 7 | {5,6} | M[r[3]] = r[2]; | r[2]:r[3]: |

(a) Combine #2 with #1, #5 with #4, and #6 with #3.

| | | | |
|---|---|---|---|
| 2 | | r[2] = M[r[30] + j]; | |
| 5 | {2} | r[2] = r[2] + 1; | |
| 6 | | r[3] = r[14] + 4; | |
| 7 | {5,6} | M[r[3]] = r[2]; | r[2]:r[3]: |

(b) Combine #7 with #6. We cannot combine #5 with #2, or #7 with #5 since we cannot combine loads/stores with arithmetic operations.

| | | | |
|---|---|---|---|
| 2 | | r[2] = M[r[30] + j]; | |
| 5 | {2} | r[2] = r[2] + 1; | |
| 7 | {5} | M[r[14] + 4] = r[2]; | r[2]: |

7. Transform the respective codes using the specified optimizations. **Also describe the benefit of performing the optimization.**

   **Answers to optimizations:** (Please find the original unoptimized code snippet in the review questions document.)

   (a) **After** Procedure integration or inlining.

```
int func1(){
  printf(``Inside func2'');
  printf(``Inside func2'');
  printf(``Inside func1'');
}

int func2(){
  printf(``Inside func2'');
}
```

   Benefits: Removes the function call/return overhead. Provides opportunities to other optimizations.

   (b) **After** Procedure specialization or cloning

```
int func1(int x, int y){
  int y;

  y = multiply(x, y);
  y = multiply16(x);
}

int multiply(int a, int b){
  return (a * b);
}

int multiply16(int a){
  return (a << 4);
}
```

   Benefit: Uses constants in function arguments (or other context information) to reduce the work done by the callee function, to improve program speed.

   (c) **After** Tail recursion elimination

```
int func1(){
  int a, b;

TOP:
  scanf(``Enter a, b: '', &a, &b);
  a = a * b;

  goto TOP;
}
```

Benefit: Removes the function call/return overhead.

(d) **After** Loop unrolling, using an unroll factor of 2

```
int func1(){
  int i, a;
  a = 0;
  for(i=0 ; i<100 ; i++){
    a += i;
    i = i + 1;
    a += i;
  }
}
```

Benefit: Reduces the number of loop branch tests and loop branches.

(e) **After** Loop collapsing

```
int func1(){
  int a[100][300];
  int *p = &a[0][0];

  for(i=0 ; i<30000 ; i++){
     *p++ = 0;
  }
}
```

Benefit: Reduces the number of loop branch tests and loop branches.

(f) **After** Loop fusion

```
int func1(){
  int a[100], b[100];

  for(i=0 ; i<100 ; i++){
    a[i] = 0;
    b[i] = 0;
  }
}
```

Benefit: Reduces the number of loop branch tests and loop branches.

(g) **After** Loop interchange

```
int func1(){
  int a[100][300];

  for(j=0 ; j<300 ; j++)
    for(i=0 ; i<100 ; i++){
       a[i][j] = 0;
  }
}
```

Benefit: Improves spatial locality of array accesses. In this case, the optimization will benefit languages that store two-dimensional arrays in column-major order in memory.

(h) **After** Scalar replacement

```
int func1(){
  int a[101], sum;

  sum = 0;
  for(i=0 ; i<100 ; i++){
    sum += a[i];
  }
  a[100] = sum;
}
```

Benefit: Scalar replacement replaces an array reference with a scalar when the index variable in a loop that does not change. Presumably the scalar will then be allocated to a register, which will improve program execution speed.

8. Answer the following:

(a) Give the advantages and disadvantages of using a three-address form of intermediate representation over a zero-address representation.

(b) What is static checking? Why is static checking preferable to dynamic checking?

(c) Describe the rules for type checking.

(d) What is coercion, overloading, and polymorphism? Give an example of each in the C language.

(e) Describe the difference between synthesized and inherited attributes. Which type of attribute does YACC support?

(f) What is the purpose of: (i) register assignment and (ii) instruction selection during code generation?

(g) What are addressing modes? When is each type typically used?
**Answer:** These are the ones we studied:
1. immediate – for constants (`r[1] = #100`),
2. register – for register operands (`r[1] = r[2]`),
3. register deferred – for pointer dereferences (`r1 = M[r[2]]`),
4. displacement – for accessing locals on the stack (`r1 = M[r[2]+100]`), and
5. indexed – for array accessed (`r1 = M[r[2]+r[3]]`).

(h) Explain the need for runtime stack management? What does it involve?

(i) What do you understand by "evaluation order of arguments"?

(j) What is code optimization ? Why is the term code *optimization* a misnomer?

(k) How are the following types of optimizations important: (i) function call optimizations, (ii) loop optimizations, (iii) memory access optimizations, (iv) control flow optimizations, (v) data flow optimizations, and (vi) machine specific optimizations.

(l) Explain each of the following optimizations in one-two lines: (i) procedure inlining (ii) tail recursion elimination (iii) loop invariant code motion (iv) loop strength reduction (v) induction variable elimination (vi) loop unrolling (vii) register allocation (Vs. register assignment)

(m) What are activation records?

(n) What are *caller-save* and *callee-save* registers?

(o) What is a *call-graph*?

(p) What are *non-local* names?

(q) Identify the fields of a general activation record in Figure 7.5 in your text-book.

(r) Describe characteristics, advantages, and disadvantages of the following data allocation strategies: (i) static allocation, (ii) stack allocation, and (iii) heap allocation.

(s) Name and describe the features of the two types of *implicit* heap storage reclamation strategies. What is its main drawback?

(t) What is an access link? How is access link different from a control link?
**Answer:** Access link is used to find nonlocal values in a block structured language. For a given function, the access link points to the latest activation of the procedure that most immediately encapsulates this function. Thus, access link is a static property of the program. In contrast, a control link points to the *caller* function, and is a dynamic property of the program.