Computation Structures
Compilation Worksheet

compile_expr(expr) ⇒ Rx

- Constants: `1234` ⇒ Rx

```
– CMOVE(1234,Rx)
– LD(c1,Rx)
  …
  c1: LONG(123456)
```

- Variables: `a` ⇒ Rx

```
– LD(a,Rx)
  . . .
  a: LONG(0)
```

- Variables: `b[expr]` ⇒ Rx

```
– compile_expr(expr)⇒Rx
  MULC(Rx,bsize,Rx)
  LD(Rx,b,Rx)
  …
  // reserve array space
  b: . = . + bsize*blen
```

- Operations: $expr_1$ `+` $expr_2$ ⇒ Rx

```
– compile_expr(expr1)⇒Rx
  compile_expr(expr2)⇒Ry
  ADD(Rx,Ry,Rx)
```

- Procedure call: `f(e1, e2, …)` ⇒ Rx

  next lecture!

- Assignment: `a=expr` ⇒ Rx

```
– compile_expr(expr)⇒Rx
  ST(Rx,a)
```

compile_statement(…)

- Unconditional: `expr;`

```
– compile_expr(expr)
```

- Compound: `{ s1; s2; … }`

```
– compile_statement(s1)
  compile_statement(s2)
  …
```

- Conditional: `if (expr) s1;`

```
– compile_expr(expr)⇒Rx
  BF(Rx,Lendif)
  compile_statement(s1)
Lendif:
```

- Conditional: `if (expr) s1; else s2;`

```
– compile_expr(expr)⇒Rx
  BF(Rx,Lelse)
  compile_statement(s1)
  BR(Lendif)
Lelse:
  compile_statement(s1)
Lendif:
```

- Iteration: while (*expr*) *s1*;

```
– BR(Ltest)
Lwhile:
  compile_statement(s1)
Ltest:
  compile_expr(expr)⇒Rx
  BT(Rx, Lwhile)
```

- Iteration: `for (init; test; incr) s1;`

```
  init;
  while (test) { s1; incr; }
```

Problem 1.

Please hand-compile the following snippets of C code into equivalent Beta assembly language statements. Assume that memory locations have been allocated for the all C variables with labels that corresponds to the variable names. So to load the value of the C variable `a` into register `R3`, the appropriate assembly language statement would be `LD(R31,a,R3)`. And to store the value in `R17` to the C variable `b`, the appropriate assembly language statement would be `ST(R17,b,R31)`. Similarly, assume that memory locations have been allocated for each C array, with a label defined whose value is the address of the 0th element of the array.

(A)
```
a = 42;
```

(B)
```
c = 5*x – 13;
```

(C)
```
y = (x – 3)*(y + 123456);
```

(D)
```
if (a == 3) b = b + 1;
```

(E)
```
a[i] = a[i-1];
```

(F)
```
x = y[3] + y[12];
```

(G)
```
if (b == 0 || b < min) {
    min = b;
} else {
    too_big += 1;
}
```

(H)
```
sum = 0;
i = 0;
while (i < 10) {
    sum = sum + i
    i = i + 1;
}
```

Problem 2.

In block-structured languages such as C or Java, the scope of a variable declared locally within a block extends only over that block, i.e., the value of the local variable cannot be accessed outside the block. Conceptually, storage is allocated for the variable when the block is entered and deallocated when the block is exited. In many cases, this means the compiler if free to use a register to hold the value of the local variable instead of a memory location.

Consider the following C fragment:

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+1) sum += i;
}
```

A. Hand-compile this loop into assembly language, using registers to hold the values of the local variables "i" and "sum".

B. Define a *memory access* as any access to memory, i.e., instruction fetch, data read (LD), or data write (ST). Compare the number of total number of memory accesses generated by executing the optimized loop with the total number of memory access for the unoptimized loop (part G of the preceding problem).

C. Some optimizing compilers "unroll" small loops to amortize the overhead of each loop iteration over more instructions in the body of the loop. For example, one unrolling of the loop above would be equivalent to rewriting the program as

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+2) {
      sum += i; sum += i+1;
  }
}
```

Hand-compile this loop into Beta assembly language and compare the total number of memory accesses generated when it executes to the total number of memory accesses from part (1).

Problem 3.

Which of the following Beta instruction sequences might have resulted from compiling the following C statement? For each sequence describe the value that does end up as the value of y.

```
int x[20], y;
y = x[1] + 4;
```

```
A. LD (R31, x + 1, R0)
   ADDC (R0, 4, R0)
   ST (R0, y, R31)
```

```
B. CMOVE (4, R0)
   ADDC (R0, x + 4, R0)
   ST (R0, y, R31)
```

```
C. LD (R31, x + 4, R0)
   ST (R0, y + 4, R31)
```

```
D. CMOVE (4, R0)
   LD (R0, x, R1)
   ST (R1, y, R0)
```

```
E. LD (R31, x + 4, R0)
   ADDC (R0, 4, R0)
   ST (R0, y, R31)
```

```
F. ADDC (R31, x + 1, R0)
   ADDC (R0, 4, R0)
   ST (R0, y, R31)
```