1. Object Layout [30 pts.]

Consider the following Java code:

```java
class A {
    int x, y;
    A(int x, int y) { this.x = x; this.y = y; }
}

void foo() {
    A[] arr = new A[3];
    for (int i = 0; i < arr.length; i++)
        arr[i] = new A(i, 10 * i);
}
```

(a) Draw a picture showing the memory layout of `arr` and its elements (including all headers) at
the point just before `foo` returns, as it would appear in the AIX/Power-PC implementation of the
Jalapeño system, according to the assigned paper by Alpern, et al.

(b) Under Linux/IA-32, reads from page 0 cause pointer faults whereas reads from very high
memory addresses do not. In light of this, how should this memory layout be changed under the
Linux/IA-32 implementation of Jalepeño (or Jikes, as it is now called).

(c) Some Java compilers use an optimization called object inlining. In the usual form of this
optimization, if the compiler is certain that an object A is pointed to only once, from another
object B, then A can be allocated as part of B’s heap record, rather than in its own, separate heap
record. This saves the cost of dereferencing the pointer to A, and the GC overheads associated with
the second record. (See Section 3.2.3 of the assigned paper by Scales et al. for more discussion
of this optimization.) Based on the ideas in the assigned paper by Chilimbi, Hill, and Larus, argue
why object inlining is not necessarily a good idea. Sketch some Java code that might be made
slower if object inlining were performed.

2. Optimizations [15 pts.]

Many program transformations used by compilers can be broadly classified into two categories.

1. Control flow elimination removes jumps, tests, and associated bookkeeping code, typically
by replicating code. It typically makes the program faster, but larger (although subsequent
application of other optimizations may shrink the program again). One example of such
an optimization is loop unrolling, which usually increases program size, but removes the
overhead of performing the loop test.
2. **Redundancy elimination** removes redundant computations. It typically make the program both smaller and faster. One example is hoisting invariant code out of loops.

Consider the performance results for the Swift compiler described in Table 2 of the assigned paper by Scales, *et al.* Based on these results, argue which category of transformations is more important for Swift. (Note: not all the optimizations described in this table fit into either category; you’ll have to decide which ones do.)

3. **Garbage collection** [15 pts.]

Suppose we are trying to choose between a mark-and-sweep collector and a copying collector for a particular system. Explain how each of the following factors, considered independently, might affect our decision (if at all). You may wish to make additional assumptions about the system; if so, be sure to state them.

(a) All records allocated by the system are the same size.

(b) On average, 99% of the allocated data is garbage at any given program point.

(c) Once allocated, records are never updated.

(d) Our system works by preprocessing a new source language into C code, which is then compiled by an existing, ordinary C compiler.

(e) Our collector needs to have bounded pause times.

4. **Program Representations** [25 pts.]

Consider the following program fragment, written in 3-address code.

```plaintext
a <- 5
b <- 10
i <- 1
goto L4
L1: if i < a goto L2
c <- i * 2
goto L3
L2: c <- i * 3
L3: a <- a + c
i <- i + 1
L4: if i < b goto L1
d <- a + b
```

(a) Identify the basic blocks and draw a control flow diagram using the basic blocks as the nodes.

(b) Number the basic blocks and draw a dominator tree for the procedure.

(c) Identify the dominance frontier of each block that contains one or more assignments.

(d) Put the procedure into SSA form (displayed as a control flow diagram).

(e) Put the procedure into Reference-Safe SSA form, as described in Section 2 of the assigned paper by Amme, *et al.*
5. Verification [15 pts.]
Consider the following bytecode sequence for a function:

```java
public static int example();
Code:
  0:    iconst_4
  1:    istore_1
  2:    iload_1
  3:    iinc   1, -1
  6:    iload_1
  7:    ifne   2
 10:    iadd
 11:    iadd
 12:    iadd
 13:    ireturn
```

(a) If this function were executed, what operations would it perform on the stack and what value would it return?

(b) This function will fail to pass the Java bytecode verifier. Why, exactly?

(c) Consider the following translation of the function into (not especially efficient) TALx86 assembly code:

```assembly
example: ∀ρ:Tss. {esp: sptr {eax: B4,esp:sptr ρ} :: ρ}
mov  ebx, 4
top: ∀ρ:Tss. {ebx: B4, esp: sptr ρ}
push ebx
dec ebx
cmp ebx, 0
jne tapp(top,<B4::ρ>)
pop eax
pop ebx
add eax, ebx
pop ebx
add eax, ebx
post ebx
add eax, ebx
retn
```

(Note: Recall that the X86 uses two-address code; the convention for this assembler is that the destination register is listed first. So, for example, each add instruction adds ebx to eax and puts the result in eax.)

Is this TALx86 function well-typed? Explain why or why not.