CIS (4|5)61
Spring 2013 Midterm 2

1. [10 points]

Consider the following grammar, which is supposed to accept expressions involving pointer de-reference and addition. For example, it should accept $i**+i*$. **S** (sum) and **L** (location) are the terminal symbols, as well as the non-terminal start symbol **P**). i (identifier) and the punctuation symbols '*' (for dereferencing a pointer) and '+' (addition) are terminal symbols, plus the end-of-file symbol $. Show that it is not LR(0), and that at least one potential LR(0) conflict is resolved by LALR(1) lookahead.

P $\rightarrow$ **S** *$*

S $\rightarrow$ **S** *+* **L**

S $\rightarrow$ **L**

L $\rightarrow$ **L** ***

L $\rightarrow$ *i*

(score)

2. [10 points] Many programming languages, including C and Java, permit using assignments as a sort of side-effect within expressions. For example, in C or Java one can write

```
x = y = z = 7;
```

to set variables x, y, and z all to 7. One often sees reuse of assigned values in loops, e.g.,

```
while (ch = getchar()) {
    // do something with ch
}
```

(where the loop is ended when `getchar( )` returns the value 0, interpreted as *false* in C).

The following type inference rule in Cool does not permit re-using values in assignments in this way:

$$[\textsc{Assign}]\ \frac{\begin{array}{c} O(v) = T \\ O, M \vdash e_1 : T' \\ T' \leq T \end{array}}{O, M \vdash v \texttt{ = } e_1 : \texttt{Unit}}$$

What small change would you make to this type rule in Cool to permit assigned values to be reused in Cool as they can be in C and Java? You can either write the modified type inference rule below, or cross out and replace part of the type inference rule above.

3. [10 points] For this question, we imagine a computer with the following instruction set:

```
LDA r_i,x      ;; The address of location x is loaded into register r
LDC r_i,k      ;; r_i := k (k is a constant)
LD r_i,x       ;; r_i := Mem[x]
ST r_i,r_k     ;; Mem[r_k] := r_i
ADD r_i,r_j,r_k ;; r_i := r_j + r_k
SUB r_i,r_j,r_k ;; r_i := r_j - r_k
BP label       ;; Branch to label if the last result was positive
BZ label       ;; Branch to label if the last result was zero
B label        ;; Branch to label (regardless of the condition)
```

Assume an unlimited set of registers, and assume that a, b, and c are integer variables that have already been allocated in the activation record. (They are like Java or C `int` variables, directly in the activation record, and not like Java `Integer` or Cool `Int` variables, which are objects wrapping integers.) Show the register code you might generate for the following (which could be Java or C or something similar with short-circuit evaluation of boolean expressions. `||` is logical *or*.)

```
if (a > b || b > c) {
   a = a + b;
}
```


(score)

4. [10 points] Suppose class Foo has an attribute (instance variable) x which is normally allocated starting at the 24th byte of a Foo object. Class Bar is a subclass of class Foo, and class Bar declares an additional attribute y. Should x also be allocated starting at the 24th byte of Bar objects? Why? Is it possible for objects of class Bar to be smaller than objects of class Foo? Why?