Advanced Programming Languages
Homework Assignment 4

Exercise 1: In class we gave the following rules for the (backward) verification condition generation of assignment and let:

$$\begin{array}{ll} \mathrm{VC}(c_1; c_2, B) & = \mathrm{VC}(c_1, \mathrm{VC}(c_2, B)) \\ \mathrm{VC}(x := e, B) & = [e/x]\ B \\ \mathrm{VC}(\mathsf{let}\ x = e\ \mathsf{in}\ c, B) & = [e/x]\ \mathrm{VC}(c, B) \end{array}$$

That rule for let has a bug. Give a correct rule for let.

Exercise 2: **Extra Credit.** Given $\{A\}c\{B\}$ we desire that $A \Rightarrow \mathrm{VC}(c, B) \Rightarrow \mathrm{WP}(c, B)$. We say that our VC rules are *sound* if $\models \{\mathrm{VC}(c, B)\}\ c\ \{B\}$. Demonstrate the unsoundness of the buggy let rule by giving the following six things:

1. a command c and
2. a post-condition B and
3. a state σ such that
4. $\sigma \models \mathrm{VC}(c, B)$ and
5. $\langle c, \sigma \rangle \Downarrow \sigma'$ but
6. $\sigma' \not\models B$.

Exercise 3: Write a sound and complete Hoare rule for do c while b. This statement has the standard semantics (e.g., c is executed at least once, before b is tested).

Exercise 4: Choose exactly *one* of the 4A and 4B below. (If you are not certain, pick 4A. The answers end up being equivalent, but 4A may be easier to grasp for some students and 4B easier to grasp for others.)

4A. Give the (backward) verification condition formula for the command $\mathsf{do}_{Inv}\ c\ \mathsf{while}\ b$ with respect to a post-condition P. The invariant Inv is true before each evaluation of the predicate b. Your answer may not be defined in terms of VC(while...).

4B. Give the (backward) verification condition formula for the command $\mathsf{do}_{Inv1,Inv2}\ c\ \mathsf{while}\ b$ with respect to a post-condition P. The invariant *Inv1* is true before c is first executed. The invariant *Inv2* is true before each evaluation of the loop predicate b. Your answer may not be defined in terms of VC(while...).

Exercise 5: Consider the following three alternate while Hoare rules (named lannister, stark, and targaryen):

$$\frac{\vdash \{X\}\ c\ \{b \Rightarrow X \wedge \neg b \Rightarrow Y\}}{\vdash \{b \Rightarrow X \wedge \neg b \Rightarrow Y\}\mathsf{while}\ b\ \mathsf{do}\ c\ \{Y\}}\ \mathsf{lannister} \qquad \frac{\vdash \{X \wedge b\}\ c\ \{X\}}{\vdash \{X\}\mathsf{while}\ b\ \mathsf{do}\ c\ \{X\}}\ \mathsf{stark}$$

$$\frac{\vdash \{X\}\ c\ \{X\}}{\vdash \{X\}\mathsf{while}\ b\ \mathsf{do}\ c\ \{X \wedge \neg b\}}\ \mathsf{targaryen}$$

All three rules are sound but incomplete. Choose two incomplete rules. For each chosen rule provide the following:

1. the name of the rule and
2. A and
3. B and
4. σ and
5. σ' and
6. c such that
7. $\langle c, \sigma\rangle \Downarrow \sigma'$ and
8. $\sigma \models A$ and
9. $\sigma' \models B$ but
10. it is not possible to prove $\vdash \{A\}\ c\ \{B\}$.

Flavor text: Incompleteness in an axiomatic semantics or type system is typically not as dire as unsoundness. An incomplete system cannot prove all possible properties or handle all possible programs. Many research results that claim to work for the C language, for example, are actually incomplete because they do not address `setjmp/longjmp` or bitfields. (Many of them are also unsound because they do not correctly model unsafe casts, pointer arithmetic, or integer overflow.)

Exercise 6: No coding component. Instead, tell me how long you spent on this, something you like about the class (or the work, or the project, or whatever), something you do not like, and something I do not know about you. All non-null answers receive full credit. Think about your project proposal.