What to turn in

Turn in the written parts of the assignment during class on the due date. For the programming part, you should mail your version of the file `verification.sml` to `bwh6@cs.cornell.edu` by 5PM on that day.

Problem 1: Hoare Rules

In this problem we slightly change the Hoare rule for the while loop and then examine if the resulting proof systems are sound and relatively complete. First, we replace the Hoare rule for while loops with the following alternative rule:

$$\frac{\{A\}c\{A\}}{\{A\}\textbf{while } b \textbf{ do } c\{A \wedge \neg b\}}$$

Prove or disprove the following claims:

(1a) The resulting proof system sound.

(1b) The resulting proof system relatively complete.

Second, we replace the Hoare while rule with the following:

$$\frac{\{A \wedge b\}c\{A \vee b\}}{\{A\}\textbf{while } b \textbf{ do } c\{A \wedge \neg b\}}$$

(1c) Argue whether the new proof system is sound and relatively complete, and prove your results.

Problem 2: Weakest Preconditions of While Loops

Consider a relation $wp'(c, B)$, which is the same as the weakest precondition wp defined in class for all constructs, except the while loop $w \equiv \textbf{while } b \textbf{ do } c$, for which we define it as an infinite disjunction:

$$\begin{aligned}
wp'(\textbf{skip}, B) &= B \\
wp'(X := a, B) &= B[X \mapsto a] \\
wp'(c_0; c_1, B) &= wp'(c_0, wp'(c_1, B)) \\
wp'(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, B) &= (b \wedge wp'(c_0, B)) \vee (\neg b \wedge wp'(c_1, B)) \\
wp'(w, B) &= \bigvee_{i \geq 0} P_i
\end{aligned}$$

The predicates P_i are inductively defined as follows:

$$\begin{aligned}
P_0 &= \neg b \wedge B \\
P_{i+1} &= b \wedge wp'(c, P_i)
\end{aligned}$$

Assume that we allow infinite disjunctions in our assertion language:

$$A := \ldots \quad | \quad \bigvee_{i \geq 0} A_i$$

The validity relation $\models A$ is defined as in class for all assertions, except for infinite disjunctions. The validity of $\bigvee_{i \geq 0} A_i$ in a state σ and under an interpretation I is defined as:

$$\sigma \models^I \bigvee_{i \geq 0} A_i \overset{def}{\iff} \exists i \geq 0 \;.\; \sigma \models^I A_i$$

2a) Denote $W = wp'(w, B)$. Show that each the following assertions are valid:

$$\begin{aligned} \neg b \wedge W &\Leftrightarrow \neg b \wedge B \\ b \wedge W &\Leftrightarrow b \wedge wp'(c, W) \end{aligned}$$

2b) Assume a definition of validity of partial correctness assertions based on operational semantics. More precisely, we say that $\{A\}c\{B\}$ is valid if and only if:

$$\forall \sigma, \sigma' \, . \, (\sigma \models A \wedge \langle c, \sigma \rangle \rightarrow \sigma') \;\Rightarrow\; \sigma' \models B$$

With this definition, show that the triple $\{wp'(c, B)\}c\{B\}$ is valid for all commands c.

2c) Is $\{wp'(w, B)\}$ a weakest precondition for the loop construct? Prove your result or show a counterexample.

Problem 3: For Loops

In this problem you are required to write the Hoare rules for two variants of for loops.

3a) Consider that we replace the IMP while loop with a for loop construct:

$$\textbf{for } (X = a_1 \textbf{ to } a_2 \textbf{ step } a_3)\ c$$

The meaning of this construct is the same as that of the Java or C statement: `for (X = a1; X <= a2; X += a3) c`. Write a Hoare rule for this construct.

3b) Consider now a simpler version of the for loop, with the following syntax:

$$\textbf{for } (X = n_1 \textbf{ to } n_2)\ c$$

Informally, this construct executes as follows. Given two numbers n_1 and n_2, if $n_1 \leq n_2$, the loop executes exactly $n_2 - n_1 + 1$ iterations. At iteration i (with $0 \leq i \leq n_2 - n_1$), it assigns value $n_1 + i$ to variable X and then executes the loop body. Similarly, if $n_1 > n_2$, it executes exactly $n_1 - n_2 + 1$ iterations. At iteration i (with $0 \leq i \leq n_1 - n_2$), it assigns value $n_1 - i$ to variable X and then executes the loop body. Write a Hoare rule for this construct.

Problem 4: Annotated Programs and Verification Conditions

In this problem we explore the generation of verification conditions for annotated programs, with the goal of automatically proving partial correctness assertions.

First consider the IMP language *extended* with for loops, as defined in problem 3b:

$$\textbf{for } (X = n_1 \textbf{ to } n_2)\ c$$

The new language has both while loops and for loops. We want to design an annotation language on top of it and use the annotations to prove partial correctness assertions for annotated programs.

As we saw in class, the process of proving partial correctness is difficult in part because it's difficult to automatically compute loop invariants. One solution is to use an annotation language where the user provides the necessary loop invariants and the compiler then checks if those invariants are valid. The annotation language that we use in this assignment is similar to the one given in class. However, there are a couple of differences:

$$
\begin{array}{rcl}
aexp & ::= & n \ \mid\ \texttt{X} \ \mid\ aexp + aexp \ \mid\ aexp * aexp \ \mid\ aexp - aexp \ \mid\ (\ aexp\ ) \\
\\
bexp & ::= & \texttt{true} \ \mid\ \texttt{false} \ \mid\ aexp = aexp \ \mid\ aexp <= aexp \ \mid\ \texttt{not}\ bexp \\
 & & \quad \mid\ bexp\ \texttt{or}\ bexp \ \mid\ bexp\ \texttt{and}\ bexp \ \mid\ bexp \Rightarrow bexp \ \mid\ (\ bexp\ ) \\
\\
assn & ::= & bexp \\
com & ::= & \texttt{skip} \ \mid\ \texttt{X} := aexp \ \mid\ com\ ;\ com \\
 & & \quad \mid\ \texttt{if}\ (\ bexp\ )\ \texttt{then}\ \{\ com\ \}\ \texttt{else}\ \{\ com\ \} \\
 & & \quad \mid\ \texttt{while}\ \{\ assn\ \}\ (\ bexp\ )\ \texttt{do}\ \{\ com\ \} \\
 & & \quad \mid\ \texttt{for}\ \{\ assn\ \}\ (\ X = n_1\ \texttt{to}\ n_2\ )\ \{\ com\ \}
\end{array}
$$

Figure 1: Concrete Syntax for Annotated IMP Programs with For Loops

- We will not use annotations for sequences of programs, because they are not really difficult to extract once we know the loop invariants.
- We write the annotations on the loop tests, with the meaning that the assertions must hold whenever the program tests the loop condition. The abstract syntax for annotated loops is therefore:

  $$\textbf{while}\ \{D\}b\ \textbf{do}\ c$$
  $$\textbf{for}\ \{D\}(X = n_1\ \textbf{to}\ n_2)\ c$$

  where $\{D\}$ the loop invariant assertion that must hold at the beginning of each iteration, and c is the loop body.
- For simplicity, we assume that the assertion language is just the language of boolean expressions **Bexp**. Therefore, there are no additional integer variables and no universal or existential quantifiers in the assertions. In this text we refer to assertions using upper-case letters and to booleans using lower-case letters.
- We allow implications in boolean expressions ($b_0 \Rightarrow b_1$) and in assertions ($A_0 \Rightarrow A_1$).

The concrete syntax of annotated IMP command with for loops and implication operators is shown in Figure 1. Unlike the previous programming assignment, we now use the keyword `do` in the while loop **while** $\{D\}b$ **do** $\{c\}$ for readability purposes, to avoid reading $\{c\}$ as a postcondition for b.

4a) **Verification Conditions**

Give a formal definition of the verification conditions for annotated IMP programs in the form of an expression $VC(\{A\}c\{B\})$, where $\{A\}$ and $\{B\}$ are assertions and c is an annotated command. You must construct the verification condition $VC(\{A\}c\{B\})$ such that its validity implies the validity of $\{A\}c_u\{B\}$, where c_u is the un-annotated version of c.

4b) **Implementation**

The code supporting this assignment provides a lexer and a parser for annotated IMP programs with for loops.

You are required to implement a verification condition generator function, `VC : Bexp * Com * Bexp` → `Bexp`, which takes the partial correctness assertion of an annotated command and produces its verification condition, according to the description you gave in (4a).

Ideally, the resulting verification condition would be fed into an automated theorem prover which would try to demonstrate whether the formula holds or not. Instead of building such a tool, you will implement a function `rewriteAssn : Bexp → Bexp` which will simplify the verification condition by repeatedly applying rewriting rules. Suggested rules are as follows:

– Algebraic and boolean identities:

$$
\begin{aligned}
&\forall a \in \mathbf{Aexp}\ .\ 0 + n = n + 0 = n \\
&\forall a \in \mathbf{Aexp}\ .\ 0 * n = n * 0 = 0 \\
&\forall a \in \mathbf{Aexp}\ .\ 1 * n = n * 1 = n \\
&\forall b \in \mathbf{Bexp}\ .\ \mathbf{false} \vee b = b \vee \mathbf{false} = b \\
&\forall b \in \mathbf{Bexp}\ .\ \mathbf{true} \vee b = b \vee \mathbf{true} = \mathbf{true} \\
&\forall b \in \mathbf{Bexp}\ .\ \mathbf{false} \wedge b = b \wedge \mathbf{false} = \mathbf{false} \\
&\forall b \in \mathbf{Bexp}\ .\ \mathbf{true} \wedge b = b \wedge \mathbf{true} = b \\
&\forall b \in \mathbf{Bexp}\ .\ \mathbf{true} \Rightarrow b = b \\
&\forall b \in \mathbf{Bexp}\ .\ \mathbf{false} \Rightarrow b = \mathbf{true} \\
&\forall b \in \mathbf{Bexp}\ .\ b \Rightarrow \mathbf{true} = \mathbf{true} \\
&\forall b \in \mathbf{Bexp}\ .\ b \Rightarrow \mathbf{false} = \neg b
\end{aligned}
$$

– Constant folding:

  * if n and n' are integer constants, rewrite the expressions $n+n'$, $n-n'$, $n*n'$ with the constant integer values they represent;
  * if n and n' are integer constants, rewrite the expressions $n = n'$ and $n \leq n'$ with the corresponding constant boolean values;
  * if t and t' are boolean constants, rewrite the expressions $t \vee t'$, $t \wedge t'$, $t \Rightarrow t'$, $\neg t$ with the constant boolean values they represent;

Feel free to add any other rewriting rule to simplify your verification conditions. Mention such additional rules in your write-up. Ideally, your verification conditions should be simplified to **true** whenever the given partial correctness assertion is valid.

You should download the code for this assignment from the file `imp-vc.tar.gz` given on the CS611 Homework web page. The top-level loop of the given code parses an input file with an annotated partial correctness assertion, invokes the verification condition generator `VC`, prints out the generated condition, then simplifies it and prints out the simplified verification condition. Your task is to implement the functions `VC` and `simplifyAssn` in the `verification.sml` file. This file is the only one you should modify.