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 `schong@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:

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

(1a) Is the resulting proof system sound? Prove your answer.

(1b) Is the resulting proof system relatively complete? As discussed in class, relative completeness refers to the fact that you get derivations for assertions for free. Again, you must prove your answer.

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

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

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

Problem 2: Do-Until Loops

As we did in homework 1, we will again examine the semantics of the **do c until b** construct, this time in the context of axiomatic semantics. Informally, at each iteration we first execute the command c and then we test the condition b. If the condition evaluates to false the execution continues with the next iteration; otherwise, it terminates. In other words, the do-until construct has the operational semantics that you were required to give in Homework 1.

We next redefine the validity of Hoare triples using the operational semantics of the command in the triple (rather than its denotational semantics, as given in class):

$$\models \{A\}c\{B\} \quad \text{iff} \quad \forall \sigma, \sigma', I \;.\; \sigma \models^I \{A\} \;\Rightarrow\; (\; \langle c, \sigma \rangle \rightarrow \sigma' \;\Rightarrow\; \sigma' \models^I \{B\} \;)$$

Note that the operational semantics of the do-until construct and the above definition describe the validity relation $\models \{A\}\textbf{do}\, c\, \textbf{until}\, b\{B\}$.

We also redefine weakest preconditions based on operational semantics, i.e. wp is the weakest precondition of command c and postcondition B if:

$$\forall \sigma, \sigma', I \;.\;\; \sigma \models^I \{wp\} \quad \text{iff} \quad (\; \langle c, \sigma \rangle \rightarrow \sigma' \;\Rightarrow\; \sigma' \models^I \{B\} \;)$$

Now consider the IMP commands, but replace the while construct with the do-until construct:

$$c ::= \textbf{skip} \;\mid\; X := a \;\mid\; c_0; c_1 \;\mid\; \textbf{if}\, b\, \textbf{then}\, c_0\, \textbf{else}\, c_1 \;\mid\; \textbf{do}\, c\, \textbf{until}\, b$$

Accordingly, consider a proof system consisting of the same rules as Hoare Logic for IMP, except for the while rule, which needs to be replaced with a new rule for the do-until construct.

(2a) Give a Hoare rule for the do-until construct such that the resulting proof system is sound.

(2b) Prove that the resulting proof system with the do-until rule from part (2a) is sound, using the definition of validity based on operational semantics.

(2c) Give an infinitary formula for the weakest precondition $wp(\mathbf{do}\, c\, \mathbf{until}\, b, B)$ of the do-until construct.

(2d) Consider the inductive definition of $wp(c, B)$ given in class for IMP, but with the definition for while replaced with the one for do-until loops from part (2c):

$$\begin{array}{rcl} wp(\mathbf{skip}, B) & = & B \\ wp(X := a, B) & = & B[X \to a] \\ wp(c_0; c_1, B) & = & wp(c_0, wp(c_1, B)) \\ wp(\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, B) & = & (b \wedge wp(c_0, B)) \vee (\neg b \wedge wp(c_1, B)) \\ wp(\mathbf{do}\, c\, \mathbf{until}\, b, B) & = & \text{from (2c)} \end{array}$$

Prove that $wp(c, B)$ inductively defined above is indeed a precondition of c and B, using the definition of weakest preconditions based on operational semantics.

(2e) It can be proven, by structural induction on c, that $\vdash \{wp(c, B)\}c\{B\}$ is a theorem in the proof system constructed with the Hoare rule from (2a). Show the proof for the inductive step when $c \equiv \mathbf{do}\, c'\, \mathbf{until}\, b$.

Problem 3: 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 do-until loops; in other words, the new language has both while loops and do-until 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:

- 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 (rather than on the commands), with the meaning that the assertions must hold whenever the program tests the loop condition. The abstract syntax for annotated loops is therefore:

  $$\mathbf{while}\, \{D\} b\, \mathbf{do}\, c$$
  $$\mathbf{do}\, c\, \mathbf{until}\, \{D\} b$$

  where $\{D\}$ the loop invariant assertion that must hold whenever b is tested, 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 ($A_0 \Rightarrow A_1$).

$$
\begin{array}{rcl}
aexp & ::= & n \mid \texttt{X} \mid aexp + aexp \mid aexp * aexp \mid aexp - aexp \mid (\ aexp\ ) \\[1em]
bexp & ::= & \texttt{true} \mid \texttt{false} \mid aexp = aexp \mid aexp <= aexp \mid \texttt{not}\ bexp \\
 & & \mid bexp\ \texttt{or}\ bexp \mid bexp\ \texttt{and}\ bexp \mid bexp \Rightarrow bexp \mid (\ bexp\ ) \\[1em]
assn & ::= & bexp \\
com & ::= & \texttt{skip} \mid \texttt{X} := aexp \mid com\ ;\ com \\
 & & \mid \texttt{if}\ (\ bexp\ )\ \texttt{then}\ \{\ com\ \}\ \texttt{else}\ \{\ com\ \} \\
 & & \mid \texttt{while}\ \{\ assn\ \}\ (\ bexp\ )\ \texttt{do}\ \{\ com\ \} \\
 & & \mid \texttt{do}\ \{\ com\ \}\ \texttt{until}\ \{\ assn\ \}\ (\ bexp\ )
\end{array}
$$

Figure 1: Concrete Syntax for Annotated IMP Programs with Do-Until Loops

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

(3a) **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. Formally prove that indeed:

$$\models VC(\{A\}c\{B\}) \Rightarrow \models \{A\}c_u\{B\}$$

(3b) **Implementation**

The code supporting this assignment provides a lexer and a parser for annotated IMP programs with do-until 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 (3a).

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 the following set of rewriting rules:

– Algebraic and boolean identities:

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

- 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. However, the assignment only requires you to implement the above rules, so the verification conditions for many annotated test programs may not simplify to **true**.

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.