Homework Assignment 3

Due: November 22, 2019 at 23:00

Total points:	100
Deliverables:	`hw3.pdf` containing typeset solutions to Problems 1–3, 5, and 8.
	`proofs.rkt` containing your proof outlines for Problem 4.
	`ivl.rkt` containing your Imp to IVL translations for Problems 6-7.
	`seqlib.dfy` containing your Dafny code and annotations for Problems 9-11.

1 Implementing Imp (10 points)

In Sections 2-3 of this assignment, you will use and implement several verification tools for Imp, the simple imperative language introduced in Lecture 11. To help you get started, we have provided an implementation of Imp in Rosette. In particular, `imp.rkt` contains an interpreter for Imp programs. The interpreter assumes the following s-expression syntax for Imp, where F^* stands for zero or more repetitions of the form F:

```
P  :=  (procedure f (x*) : (x*) S)
S  :=  (skip) | (abort) | (:= x E) | (if C S S) | (while C S) | (begin S*)
C  :=  true | false | (! C) | (&& C*) | (|| C*) | (=> C C) | (<=> C C) |
       (= E E) | (<= E E) | (>= E E) | (< E E) | (> E E)
E  :=  x | Z | (+ E*) | (* E*) | (- E E*)
Z  :=  integer literal
x  :=  variable identifier
f  :=  procedure identifier
```

The file `examples.rkt` shows how to run the Imp interpreter on a few sample programs.

To implement and use tools for Imp, you will need to understand the provided semantics in detail. To help with this, study the code in `imp.rkt` and answer the following questions.

1. (2 points) The Imp grammar uses the **begin** form to compose zero or more Imp statements into a single statement. What is the meaning of the statements `(begin)` and `(begin S)` for any S?
2. (4 points) The Imp interpreter requires every variable to be assigned before it is referenced, and it also implements simple lexical scoping. Some Imp statements create local scopes, and variables defined (i.e., assigned only) in a local scope cannot be accessed outside of it. Which Imp statements (**if**, **while**, or **begin**) introduce local scopes and what part of the statement has its own local scope?
3. (4 points) What are the possible outcomes of interpreting an arbitrary Imp program on arbitrary inputs? Assume that the program is syntactically well-formed; it obeys the Imp scoping rules; and it is given the right number of inputs according to its signature.

2 Hoare Logic for Imp (20 points)

The file `hoare.rkt` contains our first verifier for Imp, and `proofs.rkt` shows how to use it to check proof outlines. This verifier implements the Hoare logic rules presented in Lecture 11, with some small optimizations that make the verifier easier to use than a literal implemenation of the rules would be.

To enable checking of Imp programs with the Hoare verifier, we modify the Imp grammar to include **#:claim** annotations that specify Hoare logic predicates to be checked:

```
S  :=  ... | (begin A*)
A  :=  S | (#:claim C)
```

To pass the Hoare verifier, a program must include enough **`#:claim`** annotations to form a valid proof outline, as defined in Lecture 11. In a complete proof outline, each statement S other than **`begin`** is surrounded by two claims, `(`**`#:claim`** P`)` S `(`**`#:claim`** Q`)`, such that $\{P\}S\{Q\}$ is a valid Hoare triple. If a claim P_1 immediately precedes a claim P_2, then it must be the case that $P_1 \implies P_2$. That is, consecutive claims must be related by the Rule of Consequence. Finally, each **`begin`** statement should start and end with claims, `(`**`begin`** `(`**`#:claim`** P`)` S^* `(`**`#:claim`** Q`))`, such that $\{P\}S^*\{Q\}$ is a valid Hoare triple.

4. (18 points) Complete the proof outlines in `proofs.rkt` so that all included calls to the verifier succeed. Your solution may add as many **`#:claim`** annotations to a program as needed. You may also wrap the bodies of procedures, **`while`** loops, and **`if`** branches with **`begin`** statements. But you may not modify the code in any other way. Submit your copy of `proofs.rkt`.

5. (2 points) The rules in `hoare.rkt` allow you to omit some annotations that would be required if the verifier used the rules given in lecture. Explain briefly (in a few sentences) how the implemented rules differ from the presented ones, and why these optimizations are correct.

3 WP, SP, and SE for Imp (30 points)

As you saw in the previous section, fully annotating programs is a lot of work! Ideally, we would write just the annotations that require human insight—preconditions, postconditions, and loop invariants—and offload the rest of the work to an automated tool. So, in this section, you will develop key parts of three such tools: a verifier based on weakest preconditions (WP), a verifier based on strongest postconditions (SP), and a bounded verifier based on symbolic execution (SE).

The code for all three tools can be found in `tools.rkt`. The tools take as input Imp programs that are annotated with preconditions, postconditions, and (optionally) loop invariants according to the following grammar, which extends the grammar from Section 1:

```
P  :=  (procedure f (x*) : (x*) (#:requires C) (#:ensures C) S)
S  :=  ... | (while C (#:invariant C) S)
```

The file `verified.rkt` contains sample programs in the extended Imp language.

Following the approach from Lecture 13, the tools do not work on this source language directly. Instead, they first transform Imp programs into loop-free code in an intermediate verification language, IVL, which drops loops and annotations from Imp and includes three new statements: `(`**`assert`** C`)`, `(`**`assume`** C`)`, and `(`**`havoc`** x`)`. Then, they compute verification conditions for the resulting program, and discharge them with Z3.

Problems 6-7 ask you to implement two different translations from Imp to IVL, one for unbounded (WP/SP) verification and one for bounded (SE) verification. The file `ivl.rkt` contains the skeleton code for both translations. When you complete the code, all tool invocations in `verified.rkt` should produce the expected results. Submit your copy of `ivl.rkt` as the answer to these problems.

Problem 8 asks you to study the code in `tools.rkt` and answer two questions about it. Include the answers to these questions in your `hw3.pdf` submission.

6. (10 points) The WP and SP verifiers expect the input program to be annotated with a precondition, postcondition, and a loop invariant for every loop in the program. Given such a program, they call the procedure `cut` from `ivl.rkt` to obtain the corresponding IVL program. Complete the implementation of `cut` to soundly eliminate all **`while`** loops as shown in Lecture 13.

7. (10 points) The SE bounded verifier expects the input program to be annotated with a precondition and a postcondition, but it requires no loop invariants. Instead, it takes as input a non-negative integer k and unrolls every loop k times to make the program finite. The unrolling transformation is performed by the procedure `unroll`. Complete the implementation of `unroll` so that the resulting IVL program

can perform up to k iterations of every loop in the original IMP program, ensuring that there are no inputs on which the transformed program fails and the original one does not. This means that `unroll` cannot insert unwinding assertions into the transformed program, as we did in Lecture 5.

8. (10 points) Study the code in `tools.rkt` and answer the following questions.

   (a) The procedure `wp` calculates the weakest precondition of **`havoc`** using a different rule than the one shown in Lecture 13. Briefly explain how the rules differ, and why the implemented one is correct.

   (b) The procedure `interpretS+` extends the IMP interpreter from `imp.rkt` to give meaning to **`assume`**, **`assert`**, and **`havoc`** statements. The SP verifier and the SE bounded verifer then use Rosette to turn this extended interpreter into a symbolic execution engine for IVL. The implementation of **`assert`** and **`havoc`** is straightforward, as it relies on the corresponding constructs provided by Rosette. But Rosette provides no construct to express assumptions. Briefly explain how `interpretS+` implements **`assume`**, and why this is a correct implementation of its semantics.

4 Verifying Programs with Dafny (40 points)

In this part of the assignment, you will use Dafny (Lecture 12) to develop a small library of verified procedures for operating on sequences. To get started, install the Visual Studio Code IDE and then follow the instructions for installing the Dafny extension for this IDE. When you first open the skeleton solution file `seqlib.dfy` in the IDE, you will see warnings and errors issued by the verifier. Once you have successfully completed the assignment, these warnings and errors will go away. Submit your copy of `seqlib.dfy` as the solution to the problems in this section.

Before you start working on the problems, you may want to read the Dafny Guide and the Sequences Tutorial. You may also want to read parts of the Dafny Reference Manual. The key constructs you will need to know for this assignment are the following: `requires`, `ensures`, `modifies`, `invariant`, and `old`.

You can solve these problems in any order you like. But we strongly recommend that you do them in the order given in the assignment and the skeleton solution. The problems ask you to write both proof annotations and code, and the comments in the solution skeleton specify what may be changed and what must remain the same. Follow the directions carefully. You may implement the missing code however you like: if it verifies, we'll take it! But making liberal use of the previously implemented components will lead to shortest solutions, with a few lines of code and annotation as opposed to potentially many. You may also find it easiest to comment out the irrelevant parts of the skeleton while working on a given problem, in order to suppress spurious warnings and make verification faster.

9. (10 points) The file `seqlib.dfy` contains a partial implementation and specification of three functions for operating on finite sequences of elements:

   - `sreverse(`s`)` returns a sequence that reverses the order of elements in the input sequence s: `sreverse([1,2,3]) == [3,2,1]`.
   - `ssubreverse(`$s, start, end$`)` returns a sequence that is like s except for reversing the subsequence between the *start* index, inclusive, and *end*, exclusive: `ssubreverse([1,2,3],0,2) == [2,1,3]`.
   - `srotate(`s, k`)` returns a sequence that concatenates the last k elements of s with the first $|s| - k$ elements of s: `srotate([1,2,3,4], 2) == [3,4,1,2]`.

   Each function is equipped with a lemma and a client procedure that tests it on specific inputs. Follow the instructions in `seqlib.dfy` to complete the implementation and specification of these functions so that their lemmas and clients are all verified. The client code and the provided annotations may not be changed, and more annotations may be added to a lemma only when stated in the comments.

10. (20 points) Now that we have a clean functional implementation of our sequence operations, we would also like to develop an efficient imperative implementation that works on arrays. The method `subreverse`, for example, uses a linear number of in-place `swap` operations to implement subsequence reversal. In fact, we specify the postcondition of `subreverse` by describing its effect on the state of the input array, when viewed as a sequence of elements. Add enough annotations to `swap` and `subreverse` so that Dafny can verify both of them, as well as as the `subreverse` client procedure. Do not change any of the provided code or annotations.

11. (10 points) As a final step, complete the array-based implementation and specification of array `reverse` and `rotate`. Both implementations should run in linear time and constant space. Add enough annotations to `reverse` and `rotate` so that Dafny can verify both of them, as well as their client procedures. Do not change any of the provided code or annotations.