Homework Assignment 4
Due: June 01, 2016 at 11:00pm

Total points: 100
Deliverables: `hw4.pdf` containing typeset solutions to Problems 2, 3, 5.
`sort.dfy` containing your Dafny implementation for Problem 1.
`hw4.smt` containing your SMT-Lib encoding for Problem 4.
`superoptimize.rkt` containing your Racket implementation for Problems 6 and 8.

1 Verifying Programs with Dafny (30 points)

In this part of the assignment, you will use Dafny (Lecture 10) to verify a modified implementation of the insertion sort. You can either download and install Dafny or use the web interface at rise4fun. To get started, read the Dafny Guide, which describes all features of Dafny that are needed to complete the assignment.

1. (30 points) sort.dfy contains an implementation of the insertion sort and a partial correctness predicate: applying the `sort` method to an array a ensures that $a[i] \leq a[j]$ for all valid indices $i < j$. This predicate is not quite right as written, however, and the implementation is missing all annotations except for the desired post-condition on `sort`.

   Get Dafny to verify sort.dfy by annotating it with sufficient pre/post conditions, assertions, loop invariants, and frame conditions. You may not change the implementation in any other way than by adding annotations. When the verification succeeds, Dafny will print the following message: "Dafny program verifier finished with n verified, 0 errors" (where n is a small number). Submit your annotated copy of sort.dfy.

   Note: This question is challenging. You might want to solve it last.

2 Symbolic Execution (30 points)

Consider the Python programs `P0` and `P1` shown below, along with a harness procedure that tests their equivalence on a given k-bit integer:

```
def P0(x, k):
  return x & -x

def P1(x, k):
  for i in range(0, k):
    if (x & (1 << i)) != 0:
      return 1 << i
  return 0

def equiv1(x, k):
  assert P0(x, k) == P1(x, k)
```

Suppose that we apply symbolic execution to evaluate `equiv1` on a symbolic k-bit integer x and a concrete positive integer k. Let $VC_1(x, k)$ denote the set of verification conditions emitted during this symbolic execution process.

2. (2 points) How many verification conditions will be generated?
3. (3 points) Can the generated verification conditions $VC_1(x, k)$ be used to *prove* that `P1` and `P0` are equivalent for a given concrete k? Explain why or why not.

4. (15 points) Encode all verification conditions from $VC_1(x, 4)$ in SMT-LIB syntax. Your encoding should list the verification conditions *in the order in which they are generated* by basic symbolic execution (i.e., depth-first, exploring 'then' branches before 'else' branches). Each verification condition should be defined using its own **`define-fun`** expression. All verification conditions should be checked individually for validity using the **`push`** / **`pop`** commands (see the SMT-LIB manual or the Z3 tutorial for details). In particular, your encoding should take the following form:

```
(declare-const x ...)
...
(define-fun vc0 ...) ; the first VC generated by symbolic execution
...
(define-fun vcn ...) ; the last VC generated by symbolic execution

(push) ; check the validity of vc0
...
(check-sat)
(pop)
...
(push) ; check the validity of vcn
...
(check-sat)
(pop)
```

   Use Z3 to check the validity of your $VC_1(x, 4)$ encoding. Report the result of running Z3 with `-st` and `-smt2` options. Submit your SMT-LIB encoding in a separate `hw4.smt` file.

5. (10 points) Consider the program `P2` shown below:

```
def P2(x, k):
  i = 0
  while ((x & (1 << i)) == 0 and i < k):
    i = i + 1
  assert P0(x, k) == x & (1 << i)
```

   The **`assert`** statement checks if `P2`'s final state is equivalent to that of `P0`. Use the technique shown in Lecture 12 to transform and annotate `P2` so that it can be used to prove the equivalence of `P2` and `P0` by emitting just three verification conditions via symbolic execution. In particular, after your transformation, symbolic execution should behave as follows when applied to `P2(x, k)` with a symbolic k-bit integer x and a concrete positive integer k:

   - it yields a set $VC_2(x, k)$ with three verification conditions whose size is independent of k, and
   - `P2` and `P0` are equivalent if and only if the verification conditions in $VC_2(x, k)$ are valid.

   Your transformed code may use the procedure `symbolic(k)` to obtain a fresh symbolic k-bit integer; it may include **`assume`** statements; and it may also use Python's **`all`** syntax.

```
# annotated and transformed code
def P2(x, k):
  ...
```

3 Program Synthesis (40 points)

Superoptimization is the task of replacing a given loop-free sequence of instructions with an equivalent sequence that is better according to some metric (e.g., shorter). Modern superoptimizers work by employing

various forms of the guess-and-check strategy: given a sequence s of instructions, they guess a better replacement sequence r, and then they check that s and r are equivalent. In Homework 2, you developed a simple SMT-based verifier for superoptimization.

In this problem, you will develop a simple program superoptimizer. Programs will be *expressions* (i.e., ASTs) with operations on 32-bit integers, together with a set of input variables. Given a program P and a grammar G from which to draw candidate replacement programs, your superoptimizer will find the shortest program in G that is equivalent to P. Because expressions are trees, we will define the "shortest" program to mean the program whose expression tree has smallest maximum depth. The only variables available in G are the input variables to P. Note that this language is much simpler than the one you built a verifier for in Homework 2.

We have provided a solution skeleton for you to complete. The solution skeleton includes a definition of the language your superoptimizer will handle (see `language.rkt`); a verifier (see `verifier.rkt`) that takes as input two programs, and returns `'EQUIVALENT` if they are equivalent or a counterexample input otherwise; and an `examples.rkt` file that demonstrates how to work with the language and the verifier. You will implement two functions in `superoptimize.rkt`. This is the only file you will submit.

You will need to install Rosette, an extension of Racket with support for solver-aided reasoning, to run the code for this problem. However, your solutions will be plain Racket—you cannot use Rosette features, and do not need to know Rosette to complete this problem. See the `README` for instructions on installing Rosette.

6. (15 points) Implement the `enumerate` function in `superoptimize.rkt`. This function takes three inputs that together define a grammar G, and iterates over all the expressions in that grammar. The function should iterate the expressions in increasing order of depth, and should not repeat expressions.

   The three inputs are:

   - `non-terminals`, a list of non-terminal elements in the grammar.
   - `terminals`, a list of terminals in the grammar.
   - `depth`, the maximum depth of the expression trees to iterate over.

   The `enumerate` function should use `in-generator` to return a lazily-generated Racket sequence, to avoid constructing all the many programs in G in memory. The `examples.rkt` file and the Racket documentation for `in-generator` provide examples of how to use generators.

   For example, invoking `enumerate` with the inputs:

   ```
   (enumerate (list bvadd bvneg) (list (variable 'x)) 2)
   ```

   should produce a sequence that iterates the following expressions:

   ```
   (variable 'x)
   (bvadd (variable 'x) (variable 'x))
   (bvneg (variable 'x))
   (bvadd (variable 'x) (bvadd (variable 'x) (variable 'x)))
   (bvadd (variable 'x) (bvneg (variable 'x)))
   (bvadd (bvadd (variable 'x) (variable 'x)) (variable 'x))
   (bvadd (bvadd (variable 'x) (variable 'x)) (bvadd (variable 'x) (variable 'x)))
   (bvadd (bvadd (variable 'x) (variable 'x)) (bvneg (variable 'x)))
   (bvadd (bvneg (variable 'x)) (variable 'x))
   (bvadd (bvneg (variable 'x)) (bvadd (variable 'x) (variable 'x)))
   (bvadd (bvneg (variable 'x)) (bvneg (variable 'x)))
   (bvneg (bvadd (variable 'x)) (variable 'x)))
   (bvneg (bvneg (variable 'x))))
   ```

7. (5 points) Identify an optimization in the above `enumerate` algorithm that, depending on the grammar, can reduce the number of programs that must be enumerated when used for superoptimization. Your optimization should not invoke a solver, and you do not need to implement your optimization.

8. (20 points) Implement the `superoptimize` function in `superoptimize.rkt`. This function takes as input a program P and a grammar G (defined in the same way as for `enumerate`), and returns the shortest program in G that is equivalent to P. If no such program exists in G, `superoptimize` should return P. The superoptimizer will use `enumerate` to iterate over programs in G.

   Run your verifier on the benchmarks in `tests.rkt` and record the outcomes in table format:

Benchmark	**Optimal Program**	**CPU Time (ms)**	**Real Time (ms)**
P1	`program`	100	200
⋮	⋮	⋮	