1 A Verifier for Superoptimization (25 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 this problem, you will develop a simple SMT-based verifier for superoptimization. Given two loop-free sequences of 32-bit integer instructions, your verifier will either confirm that they are equivalent or, if they are not, it will produce a concrete counterexample—an input on which the two sequences produce different outputs.

The verifier will accept programs in the `BV` language, which has the following grammar:

```
Prog       :=  (define-fragment (id id*) Stmt* Ret)
Stmt       :=  (define id Expr) | (set! id Expr)
Ret        :=  (return Expr)
Expr       :=  id | const | (if Expr Expr Expr) | (unary-op Expr) |
               (binary-op Expr Expr) | (nary-op Expr+)
unary-op   :=  bvneg | bvnot
binary-op  :=  = | bvule | bvult | bvuge | bvugt | bvsle | bvslt | bvsge | bvsgt |
               bvsdiv | bvsrem | bvshl | bvlshr | bvashr | bvsub
nary-op    :=  bvor | bvand | bvxor | bvadd | bvmul
id         :=  identifier
const      :=  32-bit integer | true | false
```

Assume the following well-formedness rules for programs, which your verifier does not need to check:

1. an identifier is not used before it is defined;
2. an identifier is not defined more than once;
3. the first sub-expression of an if-expression is of type boolean, and its remaining subexpressions have the same type.

The statement `(set! id Expr)` assigns the value of `Expr` to the variable `id`; the types of `id` and `Expr` must match. The inputs to a fragment are 32-bit integers.

The operators in the `BV` language have the same semantics as the corresponding operators in T_{bv} (see the Z3 tutorial on bitvectors). For example, these are valid `BV` programs:

```
(define-fragment (P1 x1 y1 x2 y2)
  (return (bvmul (bvadd x1 y1) (bvadd x2 y2))))

(define-fragment (P2 x1 y1 x2 y2)
  (define u1 (bvadd x1 y1))
  (define u2 (bvadd x2 y2))
  (return (bvmul u1 u2)))
```

1. (25 points) Implement a BMC-style verifier for the `BV` language in Racket, using the solution skeleton in the `bvv` directory. Your verifier (see `verifier.rkt`) should take as input two `BV` program fragments (see `examples.rkt` and `bv.rkt`); produce a `QF_BV` formula that is unsatisfiable iff the programs are equivalent; invoke Z3 on the generated formula (`solver.rkt`); and decode Z3's output as follows. If the programs are equivalent, the verifier should return `'EQUIVALENT`; otherwise it should return an input, expressed as a list of integers, on which the fragments produce a different output.

Inputs to the two programs should be the only unknowns (i.e., bitvector constants) in the `QF_BV` formula produced by your verifier. This means that the verifier cannot use additional constants to represent the values of program expressions and statements. But it should also not inline the translations of individual expressions. For example, consider the following `BV` fragment:

```
(define-fragment (toy b c)
  (define a (bvmul b c))
  (return (bvadd a a)))
```

The encoding may introduce two unknowns to represent the input variables `b` and `c`. But it may not translate the first statement by emitting an SMT-LIB equality assertion such as (assert $(= a$ (bvmul b c))), where a is a fresh unknown. Similarly, it may not translate the return statement by inlining the encoding of the first statement, i.e., (bvadd (bvmul b c) (bvmul b c)).

(**Hint:** Your encoding may use SMT-LIB definitions, introduced by `define-fun`.)

Your encoding should be entirely contained in the `verifier.rkt` file. In particular, the `verify-all` procedure in `tests.rkt` should be executable just by placing your `verifier.rkt` into the `bvv` directory, without modifying any supporting files. Your encoding will be tested and graded automatically, so it is important for the implementation to be self-contained, and to adhere to the input/output specification given above. Note also that we will test your code on benchmarks that are not included in `tests.rkt`. To make sure that your verifier works correctly, you will need to write additional tests of your own, especially for corner cases.