CS 270 Lab 10 (Arithmetic Simplification – programs and proofs)

Week 11 - Dec 4 – Dec 8, 2017.

Name 1:____

Drexel Username 1:____

Name 2:____

Drexel Username 2:____

Name 3:____

Drexel Username 3:____

Instructions: For this exercise you are encouraged to work in groups of two or three so that you can discuss the problems, help each other when you get stuck and check your partners work. This lab studies a recursive function to process arithmetic expressions, a recursive defined data structure which has four cases: ArithExpr := constant | variable | (+ ArithExpr ArithExpr) | (* ArithExpr ArithExpr).

The function (arith-simp expr) takes as input an arithmetic expression and returns an arithmetic expression that is equivalent to the original expression and is simplified. Two expressions are equivalent if they return the same value for all possible bindings of the variables occurring in the expressions. An expression is simplified if 1) it contains no constant arithmetic, 2) no zeros and 3) no multiplications by 1. The following rules are used to perform the simplification. They are coded in the helper routines plus-simp and mult-simp. The function arith-simp, recursively traverses the arithmetic expression and uses plus-simp and mult-simp to perform the actual simplification.

- (+ constant1 constant2) is replaced by constant1 + constant2
- (* constant1 constant2) is replaced by constant1 * constant2
- (+ expr 0) = expr = (+ 0 expr)
- (* 0 expr) = 0 = (* expr 0)
- (* 1 expr) = expr = (* expr 1)

```
; Input: (and (arith-expr? expr1) (arith-expr? expr2))
; Output: (arith-expr? (plus-simp expr1 expr2)) which is equivalent to expr
(define (plus-simp expr1 expr2)
  (cond
   [ (and (constant? expr1) (constant? expr2)) (+ expr1 expr2) ]
   [ (equal? expr1 0) expr2 ]
   [ (equal? expr2 0) expr1 ]
   [ (make-plus expr1 expr2) ]
  ) )
```

```
; Input: (and (arith-expr? expr1) (arith-expr? expr2))
; Output: (arith-expr? (mult-simp expr1 expr2)) which is equivalent to expr
(define (mult-simp expr1 expr2)
  (cond
   [ (and (constant? expr1) (constant? expr2)) (* expr1 expr2) ]
   [ (equal? expr1 0) 0 ]
   [ (equal? expr2 0) 0 ]
   [ (equal? expr1 1) expr2 ]
   [ (equal? expr2 1) expr1 ]
   [ else (make-mult expr1 expr2) ]
  )
)

; Input: (arith-expr? expr)
; Output: (arith-expr? (arith-simp expr))equivalent to expr
(define (arith-simp expr)
  (cond
    [ (constant? expr) expr ]
    [ (variable? expr) expr ]
    [ (plus? expr) (let ( [simpexpr1 (arith-simp (op1 expr))]  [simpexpr2 (arith-simp (op2 expr))] )
                             (plus-simp simpexpr1 simpexpr2)) ]
    [ (mult? expr) (let ( [simpexpr1 (arith-simp (op1 expr))]  [simpexpr2 (arith-simp (op2 expr))] )
                             (mult-simp simpexpr1 simpexpr2)) ]
  ))
```

1. Study and make sure you understand arith-simp.
2. Complete the proof, given in the lecture on Proving Properties of Recursive Functions and Data Structures, that (arith-eval (arith-simp expr) env) = (arith-eval expr env). You need to handle the case when expr = (* E1 E2).
3. Prove by induction that (is-simplified? (arith-simp expr)) is true, where

   ```
   (define (is-simplified? expr)
    (if (constant? expr)
        #t
        (and (noconstant-arith? expr) (nozeros? expr) (nomult1? expr))))
   ```