Problem 1.

Specify new operations as follows:

```
For an initial environment ρ,


minus: accepts a number and returns its negation (also a number)

(value-of (minus-exp exp) ρ)
= (num-val (- (expval->num (value-of exp ρ))))


double: accepts a number and returns its double (also a number)

(value-of (double-exp exp) ρ)
= (num-val (* 2 (expval->num (value-of exp ρ))))


gcd: accepts 2 numbers, and returns their gcd (also a number)

(value-of (gcd-exp exp1 exp2) ρ)
= (num-val (calc-gcd (expval->num (value-of exp1 ρ))
                     (expval->num (value-of exp2 ρ))))


prime?: accepts a number and tests if it's a prime (a boolean)

(value-of (prime?-exp exp) ρ)
= (bool-val (test-prime (expval->num (value-of exp ρ))))


"calc-gcd" => simple recursive gcd calculator (the euclidean algorithm)
"test-prime" => reasonably fast, iterative prime tester (slower than
fermat test, but simpler code-wise)
```

Extend language grammar in “define-the-grammar” in lang.rkt as follows:

```
; example: minus(-6) => (minus-exp <<-6>>)
(expression
     ("minus" "(" expression ")")
     minus-exp)

; example: double(3) => (double-exp <<3>>)
(expression
     ("double" "(" expression ")")
     double-exp)

; example: gcd(10,3) => (gcd-exp <<10>> <<3>>)
(expression
     ("gcd" "(" expression "," expression ")")
     gcd-exp)

; example: prime?(11) => (prime?-exp <<11>>)
(expression
     ("prime?" "(" expression ")")
     prime?-exp)
```

Extend the interpreter by adding to “value-of” in interp.rkt as follows:

```
(minus-exp (exp)
            (let ((val (value-of exp env)))
              (let ((num (expval->num val)))
                (num-val (- num)))))

(double-exp (exp)
            (let ((val (value-of exp env)))
              (let ((num (expval->num val)))
                (num-val (* 2 num)))))

(gcd-exp (exp1 exp2)
           (let ((val1 (value-of exp1 env))
                 (val2 (value-of exp2 env)))
             (let ((num1 (expval->num val1))
                   (num2 (expval->num val2)))
               (num-val
                (calc-gcd num1 num2)))))

(prime?-exp (exp)
            (let ((val (value-of exp env)))
              (let ((num (expval->num val)))
                (bool-val (test-prime num)))))
```