Problem 1 - Write a function – Remove my neighbor

(define (rm\_ngbr [lst : (Listof Number)] [value : Number]) : (Listof Number)

…)

;Tests / Outputs

(define lst '(1 2 3 4 5))

(rm\_ngbr lst 2) ;'(2 4 5)

(rm\_ngbr lst 4) ;'(1 2 4)

Problem 2 – Higher order functions for the given list, what is the output

1. (define lst2 '(1 2 3 4 5))
2. (foldl + 0 lst2 )
3. (map (λ(x) (\* x 2)) lst2)
4. (filter (λ(x) (> x 3)) lst2)
5. (filter (λ(x) (> x 3)) (map (λ(x) (\* x 2)) lst2))
6. (foldr cons '() (range 10))
7. (foldl cons '() (range 10))

Problem 3 – What is the output?

(define (g alst)

(cond

[(empty? (rest alst)) empty]

[else (if (h (length alst))

(cons (first alst) (g (rest alst)))

(g (rest alst)))]))

(define (h n)

(let ([r (remainder n 2)])

(if (zero? r) #t #f)))

(define lst2 '(11 22 33 44 55))

(g lst2)

Problem 4 – Sketch the environment

aka what is the trace of the env, show how it grows and shrinks throughout computation

1. (run `(let ([x 7]) (+ x x)))

Trace: (list (bind x 7)), and the output (numV 14)

1. (run `(let ([x 7]) (+ 4 ((lambda (y) (\* 3 y)) x))))

Trace: (list (y 7) (x 7)), and the output (numV 25)

1. (run `((lambda (x) (+ 3 ((lambda (x) (\* 2 ((lambda (x) (+ 5 x)) 7)))11)))13))

Trace: (list (x 7) (x 11) (x 13)), and the output (numV 27)

1. (run `{(lambda (x) x) (lambda (y) y)})

Problem 5 – What is the output value for the above computations?

#lang plait

(define-type Value

(numV [n : Number])

(closV [arg : Symbol]

[body : Exp]

[env : Env]))

(define-type Exp

(numE [n : Number])

(idE [s : Symbol])

(plusE [l : Exp]

[r : Exp])

(multE [l : Exp]

[r : Exp])

(lamE [n : Symbol]

[body : Exp])

(appE [fun : Exp]

[arg : Exp]))

(define-type Binding

(bind [name : Symbol]

[val : Value]))

(define-type-alias Env (Listof Binding))

(define mt-env empty)

(define extend-env cons)

(module+ test

(print-only-errors #t))

(trace extend-env)

;; parse ----------------------------------------

(define (parse [s : S-Exp]) : Exp

(cond

[(s-exp-match? `NUMBER s) (numE (s-exp->number s))]

[(s-exp-match? `SYMBOL s) (idE (s-exp->symbol s))]

[(s-exp-match? `{+ ANY ANY} s)

(plusE (parse (second (s-exp->list s)))

(parse (third (s-exp->list s))))]

[(s-exp-match? `{\* ANY ANY} s)

(multE (parse (second (s-exp->list s)))

(parse (third (s-exp->list s))))]

[(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)

(let ([bs (s-exp->list (first

(s-exp->list (second

(s-exp->list s)))))])

(appE (lamE (s-exp->symbol (first bs))

(parse (third (s-exp->list s))))

(parse (second bs))))]

[(s-exp-match? `{lambda {SYMBOL} ANY} s)

(lamE (s-exp->symbol (first (s-exp->list

(second (s-exp->list s)))))

(parse (third (s-exp->list s))))]

[(s-exp-match? `{ANY ANY} s)

(appE (parse (first (s-exp->list s)))

(parse (second (s-exp->list s))))]

[else (error 'parse "invalid input")]))

;; interp ----------------------------------------

(define (interp [a : Exp] [env : Env]) : Value

(type-case Exp a

[(numE n) (numV n)]

[(idE s) (lookup s env)]

[(plusE l r) (num+ (interp l env) (interp r env))]

[(multE l r) (num\* (interp l env) (interp r env))]

[(lamE n body) (closV n body env)]

[(appE fun arg) (type-case Value (interp fun env)

[(closV n body c-env)

(interp body

(extend-env

(bind n

(interp arg env))

c-env))]

[else (error 'interp "not a function")])]))

;; num+ and num\* ----------------------------------------

(define (num-op [op : (Number Number -> Number)] [l : Value] [r : Value]) : Value

(cond [(and (numV? l) (numV? r))

(numV (op (numV-n l) (numV-n r)))]

[else

(error 'interp "not a number")]))

(define (num+ [l : Value] [r : Value]) : Value

(num-op + l r))

(define (num\* [l : Value] [r : Value]) : Value

(num-op \* l r))

;; lookup ----------------------------------------

(define (lookup [n : Symbol] [env : Env]) : Value

(type-case (Listof Binding) env

[empty (error 'lookup "free variable")]

[(cons b rst-env) (cond

[(symbol=? n (bind-name b))

(bind-val b)]

[else (lookup n rst-env)])]))

(trace extend-env)

(define run (λ(x)(interp (parse x) mt-env ) ))

(run `(let ([x 7])

(+ 4

((lambda (y) (\* 3 y)) x))))