Part A.

The 5 components of the language:

1. Program text
2. Front end
3. Syntax tree
4. Interpreter
5. Answer

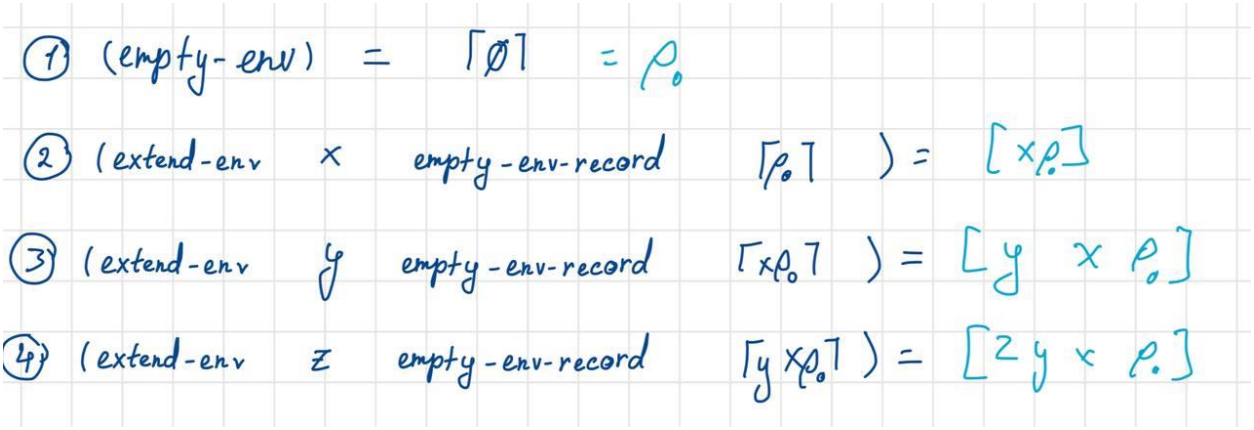
The first 3 components (Program Text, Front End, and Syntax tree) correspond to lang,rkt, interpret.rkt has the code for Interpreter, and Answer doesnt match any of the files.

Part B.

(1) [in environments.rkt]

```
(define init-env
  (lambda ()
     (extend-env
      'v (num-val 3)
      (extend-env
       'z (num-val 3)
       (extend-env
        'y (num-val 2)
        (extend-env
         'x (num-val 4)
         (empty-env))))))
```

(2)



(3)

Procedural representation – each environment is a function which takes argument, value, and the old environment.

Part C.

Denoted values -> the set of values that can be assigned to variables in the language.

Expressed values -> the set of possible values expressions can take.

In MYLET:

- Denoted values = Numbers + Lists + Boolean
- Expressed values = Numbers + Lists + Boolean

Part D.

(1)

Step 1: add the list-exp to interp.rkt in value-of part

```
(list-exp (list-val empty-list))
```

Step 2: add it to lang.rkt

```
(expression
     ("create-new-list()")
     list-exp)
```

Step 3: we need to add “empty-list” to the data-structures.rkt – the reason why I am doing it is to create an extra layer of abstraction, since the definition of the empty-list might change in racket in the future, this way we will only change the definition of empty-list in data-structures.

```
(define empty-list '())
```

Step 4: we need to add list-val to the definition of expval

```
(list-val
   (list list?))
```

Step 5: and now a converter from expval to list

```
;; expval->lst : ExpVal -> List

(define expval->list
  (lambda (v)
    (cases expval v
      (list-val (lst) lst)
      (else (expval-extractor-error 'list v)))))
```

(2)

Step 1: add the cons-exp to interp.rkt in value-of part

```
(cons-exp (exp1 exp2)
                (let ((val1 (value-of exp1 env))
                      (val2 (value-of exp2 env)))
                  (let ((num1 (expval->num val1))
                        (lst1 (expval->list val2)))
                    (if (number? num1)
                        (list-val (cons num1 lst1))
                        (list-val lst1))
                    )
                  )
                )
```

Step 2: add it to lang.rkt

```
(expression
     ("cons" expression "to" expression)
     cons-exp)
```

(3)

Step 1:

Defining mul-exp in value-of:

```
(mul-exp (exp1)
               (let ((val1 (value-of exp1 env)))
                 (let ((lst1 (expval->list val1)))
                   (num-val (mul-help lst1))
                   )))
```

Define mul-help outside of value-of:

```
(define mul-help (lambda (lst)
                       (if (null? lst)
                           0
                           (* (car lst) (if (null? (cdr lst))
                                            1
                                            (mul-help (cdr lst)))
                              ))))
```

Step2: add it to lang.rkt

```
(expression
     ("multiplication" "(" expression ")")
     mul-exp)
```

(4)

Step 1:

Defining min-exp in value-of:

```
(min-exp (exp1)
               (let ((val1 (value-of exp1 env)))
                 (let ((lst1 (expval->list val1)))
                   (if (null? lst1)
                       (num-val -1)
                       (num-val (min-help lst1 (car lst1)))
                       )
                   )))
```

Define min-help outside of value-of:

```
(define min-help (lambda (lst currentmin)
                   (if (null? lst)
                       currentmin
                       (if (<= (car lst) currentmin)
                           (min-help (cdr lst) (car lst))
                           (min-help (cdr lst) currentmin))
                       )))
```

Step2: add it to lang.rkt

```
(expression
     ("min" "(" expression ")")
     min-exp)
```

(5)

Step 1: add the if-elif-exp to interp.rkt in value-of part

```
(if-elif-exp (exp11 exp12 exp21 exp22 exp3)
                    (let ((val11 (value-of exp11 env))
                          (val12 (value-of exp12 env))
                          (val21 (value-of exp21 env))
                          (val22 (value-of exp22 env))
                          (val3 (value-of exp3 env)))
                      (cond ((expval->bool val11)
                             val12)
                            ((expval->bool val21)
                             val22)
                            (else val3))
                      ))
```

Step 2: add it to lang.rkt

```
(expression
     ("if" expression "then" expression "elif" expression "then" expression "else"
expression)
     if-elif-exp)
```

(6)

Step 1: add the rational-exp to interp.rkt in value-of part

```
(rational-exp (num1 num2)
                    (if (= 0 num2)
                        eopl:error
                        (rational-val (cons num1 num2))
                        ))
```

Step2: add it to lang.rkt

```
(expression
     ("(" number "/" number ")")
     rational-exp)
```

Step 3: add to datastructures

```
(rational-val
   (value pair?))


(define expval->rational
  (lambda (v)
    (cases expval v
      (rational-val (rational) rational)
      (num-val (num) num)
      (else (expval-extractor-error 'list v)))))

(rational-exp (num1 num2)
                    (if (= 0 num2)
                        (eopl:error 'rational-exp "Denominator cannot be zero")
                        (rational-val (cons num1 num2))
                        ))
```

(7)

Step 1: interp.rkt

```
(op-exp (exp1 exp2 op)
             (let ((val1 (value-of exp1 env))
                   (val2 (value-of exp2 env)))
                 (let ((num1 (expval->rational val1))
                       (num2 (expval->rational val2)))
                     (cond
                       ((and (number? num1) (number? num2))
                         (num-val
                           (cond
                             ((= op 1) (+ num1 num2))
                             ((= op 2) (* num1 num2))
                                   ;; -----------------------
                                   ;; INSERT YOUR CODE HERE
                                   ;; -----------------------
                             ((= op 3) (/ num1 num2))
                             (else (- num1 num2))
                                   ;; -----------------------
                             )))

                       ((and (number? num1) (not (number? num2)))
                         (rational-val
                         (let ((num2top (car num2))
                               (num2bot (cdr num2)))
                           (cond
                             ((= op 1) (cons (+ (* num1 num2bot) num2top) num2bot))
                             ((= op 2) (cons (* num1 num2top) num2bot))
                             ;; -----------------------
                             ;; INSERT YOUR CODE HERE
                             ;; -----------------------
                             ((= op 3) (cons (* num1 num2bot) num2top))
                             (else (cons (- (* num1 num2bot) num2top) num2bot))


                             ;; -----------------------


                             ))))

                       ((and (number? num2) (not (number? num1)))
                         (rational-val
                         (let ((num1top (car num1))
                               (num1bot (cdr num1)))
                           (cond
                             ((= op 1) (cons (+ (* num1bot num2) num1top) num1bot))
                             ((= op 2) (cons (* num1top num2) num1bot))
                             ;; -----------------------
                             ;; INSERT YOUR CODE HERE
                             ;; -----------------------
                             ((= op 3) (cons num1top (* num2 num1bot)))
                             (else (cons (- num1top (* num2 num1bot)) num1bot))
                             ;; -----------------------
                             ))))

                       (else
                         (rational-val
                         (let ((num1top (car num1))
                               (num1bot (cdr num1))
                               (num2top (car num2))
                               (num2bot (cdr num2)))
                           (cond
                             ((= op 1) (cons (+ (* num1top num2bot) (* num1bot num2top)) (* num1bot num2bot))) ;; add
                             ((= op 2) (cons (* num1top num2top) (* num1bot num2bot))) ;; multiply
                             ;; -----------------------
                             ;; INSERT YOUR CODE HERE
                             ;; -----------------------
                             ((= op 3) (cons (* num1top num2bot) (* num1bot num2top)))
                             (else (cons (- (* num1top num2bot) (* num2top num1bot)) (* num1bot num2bot)))


                             ;; -----------------------
                           ))))))))
```

Step 2: lang.rkt

```
(expression
     ("op(" expression "," expression "," number ")")
     op-exp)
```

(8)

Step 1:

```
(simpl-exp (input)
                  (let ((val1 (value-of input env)))
                    (let ((num1 (expval->rational val1)))
                      (cond ((= (car num1) (cdr num1))
                             (num-val 1))

                            ;just for the sake of (10 . 1) test
                            ((and (= (cdr num1) (gcd (car num1) (cdr num1)))
                                  (not (= 1 (gcd (car num1) (cdr num1)))))
                             (rational-val (cons
                              (/ (car num1) (gcd (car num1) (cdr num1)))
                              (/ (cdr num1) (gcd (car num1) (cdr num1))))))

                            ((= (cdr num1) (gcd (car num1) (cdr num1)))
                             (num-val (/ (car num1) (cdr num1))))
                            (else (rational-val (cons
                                   (/ (car num1) (gcd (car num1) (cdr num1)))
                                   (/ (cdr num1) (gcd (car num1) (cdr num1))))))))
                      )))
```

Step 2: lang.rkt

```
(expression
     ("simpl(" expression ")")
     simpl-exp)
```