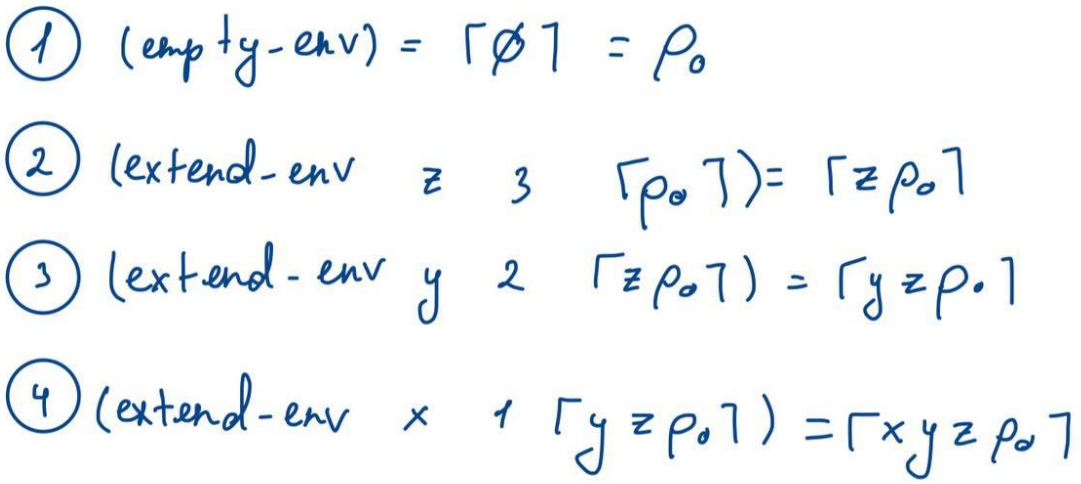
Part A.

(1)

```
(define init-env
  (lambda ()
    (extend-env
     'x (num-val 10)
     (extend-env
      'y (num-val 2)
      (extend-env
       'z (num-val 3)
       (empty-env))))))
```

(2)



Part B.

Denoted Values = Number + Boolean + Proc + Stack

Expressed Values = Number + Boolean + Proc + Stack

Part C.

(1)

Add the following to the expval definition in datatypes.rkt:

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

Add the following to the convert back from expval in datatypes.rkt:

```
;; expval->proc : ExpVal -> Stack
(define expval->stack
  (lambda (v)
    (cases expval v
      (stack-val (stack) stack)
      (else (expval-extractor-error 'proc v)))))
```

Then we need to add the stack-exp to the interpreter:

```
(stack-exp () (stack-val '()))
```

And finally, add it to the lang as a variant of expressions in our language:

```
(expression
     ("empty-stack()")
     stack-exp)
```

(2)

First let us add stack-push-exp to the interpreter:

```
(stack-push-exp (exp1 exp2)
                      (let ((val1 (value-of exp1 env))
                            (val2 (value-of exp2 env)))
                        (let ((stack (expval->stack val1))
                              (num (expval->num val2)))
                          (stack-val (cons num stack))
                          )))
```

Then we need to add it to the lang as a variant of expressions in our language:

```
(expression
     ("stack-push(" expression expression ")")
     stack-push-exp)
```

(3)

Let us add stack-pop-exp to the interpreter:

```
(stack-pop-exp (exp)
                     (let ((val (value-of exp env)))
                       (let ((stack (expval->stack val)))
                         (cond ((null? stack)
                                (display "Empty stack cannot be popped!")
                                (stack-val '()))
                               (else
                                (stack-val (cdr stack)))
                               ))))
```

Then we need to add it to the lang as a variant of expressions in our language:

```
(expression
     ("stack-pop(" expression ")")
     stack-pop-exp)
```

(4)

Let us add stack-peek-exp to the interpreter:

```
(stack-peek-exp (exp)
                      (let ((val (value-of exp env)))
                       (let ((stack (expval->stack val)))
                         (cond ((null? stack)
                                (display "Empty stack cannot be peeked!")
                                (num-val 2813))
                               (else
                                (num-val (car stack)))
                               ))))
```

Then we need to add it to the lang as a variant of expressions in our language:

```
(expression
     ("stack-peek(" expression ")")
     stack-peek-exp)
```

(5)

Let us add stack-push-multi-exp to the interpreter:

```
(stack-push-multi-exp (exp exps)
                      (let ((stack (expval->stack (value-of exp env))))
                        (let loop ((expressions exps)
                                   (current-stack stack))
                          (if (null? expressions)
                              (stack-val current-stack)
                              (let ((val (value-of (car expressions) env)))
                                (loop (cdr expressions) (cons (expval->num val) current-stack)))))))
```

Then we need to add it to the lang as a variant of expressions in our language:

```
(expression
     ("stack-push-multi(" expression (arbno "," expression) ")")
     stack-push-multi-exp)
```

(6)

Let us add stack-pop-multi-exp to the interpreter:

```
(stack-pop-multi-exp (exp1 exp2)
                          (let ((val1 (value-of exp1 env))
                                (val2 (value-of exp2 env)))
                            (let ((stack (expval->stack val1))
                                  (num (expval->num val2)))
                              (cond ((null? stack)
                                     (display "Empty stack cannot be popped")
                                     (stack-val '()))
                                    ((= num 0)
                                     (stack-val stack))
                                    (else
                                     (let loop ((remaining num) (current-stack stack))
                                       (if (or (<= remaining 0) (null? current-stack))
                                           (stack-val current-stack)
                                           (loop (- remaining 1) (cdr current-stack))))))
                              )))
```

Then we need to add it to the lang as a variant of expressions in our language:

```
(expression
     ("stack-pop-multi(" expression "," number ")")
     stack-pop-multi-exp)
```

(7)

Let us add stack-merge-exp to the interpreter:

```
(stack-merge-exp (exp1 exp2)
                      (let ((val1 (value-of exp1 env))
                            (val2 (value-of exp2 env)))
                        (let ((stack1 (expval->stack val1))
                              (stack2 (expval->stack val2)))
                          (stack-val (merge-help stack1 stack2))
                          )))
```

Helper function:

```
(define merge-help (lambda (s1 s2)
                     (append (reverse s2) s1)
                     ))
```

Then we need to add it to the lang as a variant of expressions in our language:

```
(expression
     ("stack-merge(" expression "," expression ")")
     stack-merge-exp)
```

Part D.

In many languages stacks are implemented through arrays or linked lists. That means that our language should have support for various data structures on its own (not by implementing them in scheme). To support different data structures efficiently we need to have good memory management – as is done in most modern languages.

With memory management, as we can allocate and free memory, we would need to define primitive datatypes allowed in our language and their properties which would be dictated by the denoted values of our language. The primitive datatypes would need to be operated on through cpu for best efficiency.

And we would need to define the minimal set of operations that could be done on those primitive datatypes and build everything up from there.

As soon as we have these tools – we are good to go.