COMP 301 - PROJECT 3

Team members:

Muhammet Eren Ozogul
Ada Yıldız
Aynur Ceren Cepni

PART A:

1)

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

2)

(a) (init-env) = [z=⌈3⌉, y=⌈2⌉, x=⌈10⌉]
[]
[x=⌈10⌉]
[y=⌈2⌉,x=⌈10⌉]
[z=⌈3⌉, y=⌈2⌉, x=⌈10⌉]

PART B:

ExpVal: Int + Bool + Proc + List<Int>
DenVal: Int + Bool + Proc + List<Int>

PART C:

The solutions are in the code.

PART D:

We used the list structure of the Scheme language and its built-in functions like cons, car and cdr. In addition to that, comparison symbols are used.

To be able to solve the built-in functions problem, we need to use a low level language and have the access to the memory addresses of the arrays, instead of scheme lists.Assuming we have the

access to these pointers of our arrays, by rearranging them; cons, car and cdr can be implemented very easily.

For example if we allocate memory for an array:

int* cell = (int*)malloc(sizeof(int) * 4);

cell[0] is going to give us the “car of the array” and cell[1] will represent the “cdr of the array”.

In a similar way, if we have the information of the size of the arrays, we may merge them:

```
int size1 = sizeof(arr1) / sizeof(arr1[0]);
int size2 = sizeof(arr2) / sizeof(arr2[0]);

int mergedSize = size1 + size2;

int* mergedArray = (int*)malloc(mergedSize * sizeof(int));
```

Here, the memory allocation and the sizeof functions are implemented in C, as a choice for “low level language instead of Scheme”. Any low level language that allows memory allocation actually would be useful for this purpose.

Also, for the comparison part, we already have what we need: the diff expression and the zero? expression. Using these expressions in MyProc, we can imitate the comparison symbols “<, >, =”. Basically:

If the result of the diff is zero => numbers are equal.

İf not => A bit level comparison is required. By checking the sign bit of the result:

For “<” If the sign bit is 1 => The result is a bool-val, #t

For “>” it is the other way around.

In Scheme, we need to still use built-in bitwise operations to be able to do that such as logand and arithmetic-shift.

(the overall code would be:

```
(define (sign-bit x)
  (if (zero? x) 0
        (if (zero? (logand x (arithmetic-shift 1 (- (integer-length x) 1))))
        0
        1)))
)
```

But again, a low level language like C will solve this problem since we may perform these operations directly.

Workload Distribution:

Part A: We focused on this question together.
Part B: We focused on this question together.
Part C: We divided this part into 3 and then combined our solutions to get the answer.
Part D: We focused on this question together.

Not: In every part, other than the assigned person, others have double checked each part and understood the question and solution. When one had difficulty, the others gave help to that person. In part c, we had issues when we combined the codes. All of us fixed it together.