COMP 301 Project 3

Baran Berkay Hökelek, 0060673
Berkay Barlas 0054512
Utku Noyan 0060717

December 27, 2020

Workload Breakdown

In this project first we met and discuss each section about how we can implement code and answer questions in Part A, B, and C. Then we divided parts according to their grades. Moreover, we gathered together while we have issues with implementations. You can see workload breakdown:

- Baran Berkay Hökelek: Part A(Array Implementation)
- Utku Noyan : Part B(Stack Implementation)
- Berkay Barlas: Part C(Queue Implementation)

After finishing the all part, we all checked and discussed these parts

Part A

Arrays in EREF are basically lists of references, and are expressed values themselves. This is implemented by adding the necessary modifications to data structures.scm, which provide the datatype for array, a checker for said datatype, a constructor and a method to convert EREF arrays into Scheme lists. Here is the code snippet that adds the array datatype (arr-val):

```
(define-datatype expval expval?
...
    (arr-val
     (arr (list-of reference?)))
    )
```

Here is the code snippet for the constructor:

```
(define array
 (lambda (len val)
  (letrec ((populate-arr
    (lambda (len val)
      (if (zero? len)
      '()
      (cons (newref val) (populate-arr (- len 1) val))))))
    (populate-arr len val))))
```

And here is the code snippet that converts an array into a Scheme list:

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

There are 3 expressions dealing with array manipulation: newarray-exp, updatearray-exp and readarray-exp. Following are the short descriptions for each expression, along with their inputs & outputs.

- **newarray-exp:** This expression takes 2 expressions as input: The size of the array to be created (S), and the value with which the array is to be filled (val). Then, it returns the array constructed with these values. Here is the code snippet:

```
(newarray-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env)) (val2 (value-of exp2 env)))
    (let ((length (expval->num val1)))
        (arr-val (array length val2)))))
```

- **updatearray-exp:** This expression takes 3 expressions as input: the array to be updated (A), the index which we want to update (i) and the new value for that index (val). Then, the array A is converted into a Scheme list, and the reference contained at the index i of A is set to point to val, by using the built-in procedure **setref!**. An important thing to note is that this expression returns *nothing*. Here is the code snippet:

```
(updatearray-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env))
        (val3 (value-of exp3 env)))
    (let ((index (expval->num val2)))
      (setref! (list-ref (expval->list val1) index) val3))))
```

- **readarray-exp:** This expression takes 2 expressions, the array A and the index i. Then, it converts A into a Scheme list, then de-references the i'th element (by using the built-in procedure **deref**) and returns the value. Here is the code snippet:

```
(readarray-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((index (expval->num val2)))
      (deref (list-ref (expval->list val1) index)))))
```

Part B

In this part, we implemented Stack with using arrays that implemented in Part A. We implemented new methods for stack: newstack() stack-push(stk, val), stack-pop(stk), stack-size(stk), stack-top(stk), empty-stack?(stk), print-stack(stk).

In newstack(), we initialized new array with the size of 1001, and for each value being -1. The assumptions here are that we know that maximum number of push operations is 1000, and the values in array is in range [1, 1000] given for us.

We implemented Stack with array as follows:

- We store the top index (hence, the size) of the stack at the first element of the underlying array (at index 0).
- The stack is updated by changing the $top\_index + 1^{st}$ element of the underlying array, and then incrementing the value of top_index by 1. (In stackpop-exp, the value at top_index is changed to -1 and the top_index is decremented by 1.)
- The top returns the element of array at index size and pop removes this element.
- We check the stack size by looking the first element of array. Since it is the size of the stack, if it's zero, then it means that the stack is empty.
- While displaying a stack; we use the format: $"(" + stack[0] + stack[1] + \ldots + stack[top\_index] + ")"$

Part C

In this part, we implemented Queue with array structre that we implemented in part A. For queue we implemented additional following methods: newqueue(), queue-push(queue, elm), queue-pop(queue), queue-size(queue), queue-top(queue), empty-queue?(queue), print-queue(queue).

In newqueue(), we initialized new array with the size of 1002, and for each value being -1. The assumptions here are that we know that maximum number of push operations is 1000, and the values in array is in range [1, 1000] given for us.

The additional 2 places for keeping the number of elements in queue(len) and the index of first element of queue (start-index).

We implemented Queue with array as follows:

- We used first element of array as being the size of Queue.
- We used second element of array as the beginning index of top element .
- The queue-push is changing the element of array at start-index+len and increments len (at index 1) by 1.
- The queue-top returns the element of array at len+start-index. If it's bigger than defined sized 1002, it returns the element at index len+start-index-1000.
- The queue-pop removes the first element of queue and increments start-index by 1 and decrements queue len by 1.
- We check the Queue size by looking the first element of array.
- While displaying Queue; we are iterating the element of array except the first and second element, with using parantheses.