Ahmet Hakan Hafif (64092)

Kerem Aksoy (64243)

COMP301 Project 4

1) We implement the project by sitting side by side.
2) All the implementations work properly and none of the test cases give an error or find a bug.

Question 3:

- Since we are adding a new data type to our language, we first need to define it appropriately in data structures. To do that, we use data-structures.scm file.

```
(define-datatype array array? ;;Done
  (arr ;We are holding the length of the array here
   (size number?)
   (first reference?)));We are holding only the first reference since they are contiguously allocated

; HINT if you need extractors, add them here
(define expval->arr ;An extractor for array
  (lambda (v)
    (cases expval v
      (arr-val (array) array)
      (else (expval-extractor-error 'array v)))))
```

- As always, grammar and behavior specification of the array procedures given us. Therefore, we just wrote an appropriate clause to match the required syntax, semantics and grammar.

```
(expression
 ("newarray" "(" expression "," expression ")")
 newarray-exp)

 (expression
 ("update-array" "(" expression "," expression "," expression ")")
 update-array-exp)

  (expression
 ("read-array" "(" expression "," expression ")")
 read-array-exp)

    (expression
 ("print-array" "(" expression ")")
 print-array-exp)
```

Note: to give an example; we expect 2 expressions in newarray grammar while the project file requires an integer and an expression. The reason being integers are actually an expressed values in our language, we chose to design new procedures in this way.

- Then, we specify the cases of array implementation in value-of, as always, at interp.scm with again given requirements by project pdf.

```
(newarray-exp (exp1 exp2)
               (let ((len (expval->num (value-of exp1 env)))
                     (val (value-of exp2 env)))
                 (newarray len val) ; A procedure which creates an array with  length and its first value
              ))

(update-array-exp (exp1 exp2 exp3)
               (let ((arr (expval->arr (value-of exp1 env))) ;The array
                     (idx (expval->num (value-of exp2 env))) ;Index of the entry which will be updated
                     (val (value-of exp3 env))) ;New value
                 (update-array arr idx val) ; A procedure which updates an array with given array index and new value
          ))


(read-array-exp (exp1 exp2)
               (let ((arr (expval->arr (value-of exp1 env))) ;The array
                     (idx (expval->num (value-of exp2 env)))) ;Index of the entry which will be updated
                   (read-array arr idx) ; A procedure which updates an array with given array index and new value
          ))


(print-array-exp (exp1)
               (let ((arr (expval->arr (value-of exp1 env)))) ;The array
                 (print-array arr)))
```

We also wrote some comments to specify behaviors of the cases and procedures.

- Up to this point, everything we did was usual requirements like previous projects. From now on, the actual implementation of procedures begins.
- First, to implement newarray, we use 1 helper method to handle the creation of an array.

```
(define newarray ;Returns an arr-val
  (lambda (len val)
    (if (> len 0)
        (let ((arrRef (newref val))) ;arrRef --> is the reference to the first element
          (begin
            (newarray-Helper (- len 1) val)
            (arr-val (arr len arrRef)))) ;Creating the array
        (display "Given length is not valid !") ; indicates an error occured when creating the array
        )))
```

- The tasks of the newarray are:
    - Creating the first reference of an array.
    - Calling helper method which recursively creates the array.
    - Using an extractor that implemented before to create an actual array.

```
(define newarray-Helper
  (lambda (n val)
    (if(= n 0)
    0
    (begin (newref val) ;Initially all the elements other than 0th have value 'null as default
           (newarray-Helper (- n 1) val)
          ))))
```

- Begin clause can execute more than 1 line of code and returns the value of the last one. Using that, we implement a recursive procedure that handles the creation of an array which have null values at the beginning like specified in the comment.

- Second, to implement updatearray, we figured out that we can basically use existing procedures such as setref!. To update the array in a constant time, we wrote a procedure such that:
    - Takes a value and index that value will store in it.
    - Finds first reference of the array, basically 0.
    - Throws an error if requested index + first reference value is larger than length of the array.
    - Basically, uses setref! To change the value of the reference, which is given index and computed as first reference + index.

```
(define update-array
  (lambda (arr idx val)
    (let ((first (arr-first arr)) ;Reference to the first element
          (len (arr-len arr))) ;Length of the array
      (if (>= idx len)
          (display "Given index is not valid !")
          (setref! (+ first idx) val))))) ;Using set ref by giving
```

- Third, implement to readarray, we use similar idea like the previous one. This time, deref operation has been used to dereference the reference of all elements in the array. Thanks to the recursion, we implement the procedure without and helper method.

```
(define read-array
  (lambda (arr idx)
    (let ((first (arr-first arr)) ;Taking the reference to the fi
          (len (arr-len arr))) ;Taking the length
      (if (>= idx len) ;Checking index is valid or not
          (display "Given index is not valid !")
          (deref (+ first idx)))))) ; Since we hold the first re
```

- For last one of the array procedures, namely printarray, we use 2 helper methods, one of them basically returns the size of the array.
- Similar to previous one, we cannot display the value of the elements in the array directly. Since our language consists of references. We first need to dereference these values to extract the data they are pointing in the store. To do that, our main procedure extracts the first reference of the array and the size of the array. Then, it calls for the helper method.

```
(define print-array
  (lambda (arr)
    (let ((first (arr-first arr)) ;Refe
          (len (arr-len arr))) ;Length
      (print-helper (len first)))))

(define arr-len
  (lambda (givenArray)
    (cases array givenArray
      (arr (size first)
           size))))
```

- Again, thanks to recursion, we basically deref the references and display them.

```
(define print-helper
  (lambda (n ref) ;n is the length , ref would b
    (if (= n 0)
        (display "End of the array")
        (begin
          (display (deref ref))
          (print-helper (- n 1) (+ ref 1))))))
```

PART B

- We are familiar with the stack data type from previous courses. It follows LIFO manner. It is basically a queue that works reversely.
- We wrote a grammar of stack and its procedures like previous task, as given in the project file.

```
(expression
    ("newstack()")
    newstack-exp)

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

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

 (expression
    ("stack-size" "(" expression ")")
    stack-size-exp)


(expression
    ("stack-top" "(" expression ")")
    stack-top-exp)

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

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

- Since we are familiar with the stack operations, it is easy to write the requirements in value-of

```
(newstack-exp ()
 (newarray 1000 -1)) ;-1 means null in the stack implementation since values can be 1-10000

(stack-push-exp (exp1 exp2)
  (let ((stack (expval->arr (value-of exp1 env))) ;Stack is simply an array
        (val (expval->num (value-of exp2 env)))) ;Taking the value which will be pushed
    (push stack val)))

(stack-pop-exp (exp1)
               (let ((stack (expval->arr (value-of exp1 env))))
                 (num-val (pop stack))))
(stack-top-exp (exp1)
               (let ((stack (expval->arr (value-of exp1 env))))
                 (num-val (top stack))))

(stack-size-exp (exp1)
                (let ((stack (expval->arr (value-of exp1 env))))
                  (num-val (stack-size stack))))

(empty-stack?-exp (exp1)
                  (let ((stack (expval->arr (value-of exp1 env))))
                    (bool-val (is-stack-empty stack))))

(print-stack-exp (exp1)
                 (let ((stack (expval->arr (value-of exp1 env))))
                    (print-stack stack)))
```

We use a helper function that finds next index (returns an error if stack is full). This helper procedure is used almost all of the stack operations.

```
;from end of the array to the begining of it.
(define find-next-idx ;If stack is full, returns -1
  (lambda (stack n)
    (let ((temp (read-array stack n))) ;Using read-array function
      (if (< n 0) -1
          (if (= temp -1)
              n
              (find-next-idx stack (- n 1))))
      )))
```

- In our implementation, if the next index is 999, that means stack is empty. We use this property to detect if a stack is empty or not by a basic observer procedure.

```
;this function returns true if the stack is empty
(define is-stack-empty
  (lambda (stack)
    (let ((temp (find-next-idx stack 999))) ;If the r
      (if (= 999 temp) #t ;Means the stack is empty
          #f
          ))))
```

- Since stack is basically an array, we can use updatearray procedure that we implement previously when we need to push an element to stack. The thing to remember is stack is inversely working compared to an array (LIFO). We, again can use helper method, find-next-idx to find the index and then basically push the value.

```
;Push operation
(define push
  (lambda (stack val)
    (let ((nextIdx (find-next-idx stack 999)))
      (update-array stack nextIdx val)))) ;Usin
```

- Pop and top procedures are similar in the sense of they both take the top of the stack. The difference is top procedure is just an observer while pop is not. We also have to be careful about stack's working mechanism, again.

```
(define pop
  (lambda (stack)
    (if (is-stack-empty stack)
        -1
        (let ((idx (+ (find-next-idx stack 999) 1))) ;;If we find where to push next, n-1 will be the top element
          (let ((element (read-array stack idx)))
            (begin
              (update-array stack idx -1) ; Since we pop it, we need to turn its value to -1
              element
              ))))))

;Top operation
(define top
  (lambda (stack)
    (if (is-stack-empty stack)
        -1
        (let ((idx (+ (find-next-idx stack 999) 1))) ;;If we find where to push next, n+1 will be the top element
          (let ((element (read-array stack idx)))
              element
              )))))
```

- Stack-size is basically the value returned by find-next-idx subtracted from the max size which is 999.

```
(define stack-size
  (lambda (stack)
    (let ((nextIdx (find-next-idx stack 999))) ;Since we fill the array from end to begining and find-next-idx returns
      (- 999 nextIdx)                          ;the next idx(if array is full -1),we can simply return 999 - nextidx
    )))
```

- To print stack, we again can use the read array procedure that is implemented previously. We can use recursion to check if displaying is done by comparing index of the top element with 999. The rest is basic usage of recursion and read array procedure.

```
;Stack printer
(define print-stack
  (lambda (stack)
    (let ((topIdx (+ (find-next-idx stack 999) 1)))
      (print-stack-helper stack topIdx))))

(define print-stack-helper
  (lambda (stack topIdx)
    (if (> topIdx 999) ;Means printing is done
        (display "End of the Stack")
        (begin
          (display (read-array stack topIdx)) ;Printing form can be changed
          (print-stack-helper stack (+ 1 topIdx)) ; Recursively calling
          ))))
```

PART C

```
(array-comprehension-exp (exp1 var exp2)
                         (let ((arr (expval->arr (value-of exp2 env)))) ;This returns an array
                           (let ((len (arr-len arr))) ;taking the length of the array
                             (begin
                               (arraycomp-helper exp1 arr var len 0 env) ;This returns nothing
                               (arr-val arr)) ;So we need to return that arr as exp val
                             )))
```

```
(define arraycomp-helper
  (lambda (body arr var len idx saved-env)
    (if (< idx len) ;Checking if we finished or not
       (let ((entryVal (read-array arr idx))) ;Taking the value of that entry (arr[i])
        (let ((val (value-of body (extend-env var entryVal saved-env))) ;Evaluating the body with that entry(arr[i])
              (nextIdx (+ 1 idx))) ;Calculating new index
          (begin
            (update-array arr idx val) ;;Updating the index with this value
            (arraycomp-helper body arr var len nextIdx saved-env) ;Recursively calling for the next i
            ))
        '() ;For the else condition, we do not need to nothing, just returning a empty list(null)
        )))
```

In this part, first we needed to define appropriate grammar inside lang.scm. After that, we also needed to add array-comprehension-exp in value-of function as we did while implementing arrays. To implement array-comprehension-exp part, we define a helper function called arraycomp-helper. It take 6 inputs such as body(which will be calculated for each element of the array), the array, the identifier(var), length of the array, index(initially 0), and the environment. In the implementation of this recursive procedure, we need a base case where if index >= length of the array, it stops the recursion by returning '() (dummy). The function simply evaluates the body with respect to the environment that have value of the array current entry. Then, it updates the entry with the evaluated value. Finally, it increments the index and calls the same function recursively.

A note: There leave many comments to explain our approach while implementing the project.