COMP 301 - Project 4 Report
Özkan Yamaner - 69740
Murat Güç - 68967
Group #10
June 1, 2022

Part A)

Array Implementation:

In our implementation, while defining a new data type array-val, we decided to create it as a reference. Like C's memory handle, array-val becomes the ref which points out the beginning ref value in store. Also, we have decided to store array length just before the array ref in the store. It increases the memory cost by one element,but it becomes usage friendly within figuring out the size and finding the tail in the store. For instance, if you want to create an array like the following (2,2,2,2,2), the store is designed as follows: (5,2,2,2,2,2). Consequently, datatype, extractor and procedure of the implementation is done in data-structures.scm according to ideas we determined for array-val.

a) **newarray-exp**: it gets the parameters of length and value, then sends the length to the store at the next available reference by using newref procedure. Secondly, creation of a new array returns the reference as array-val according to our implementation. So, the procedure creates a new reference for the array-value by sending the value parameter to the store. Rest of the array is constituted by a helper function which basically creates new references in the store for values recursively. Finally, newarray-exp returns array-val which points out the beginning reference of the array.

b) **read-array-exp**: This procedure takes the parameters of array and index, then returns the value. To implement that we used a helper function which is called search. It gets array-ref and index from expression after extraction, and basically deref the position at the specific index. So, summation of array-ref and index is enough to accomplish that.

c) **update-array-exp**: Update process is similar to the read-array expression, but this time we have used the setref! procedure to change val at specific index. Highlighted line is as follows: "(setref! (+ array-ref index) val)". First parameter is the reference which can be reached by the summation of array-val and index.

d) **length-array-exp**: At the beginning of the array implementation part, we mentioned that length of the array will be stored just the previous reference value of array-val. So, reaching the -1th index will accomplish our purpose. To do that, we have used the search helper function again like the following: (search array-ref -1).

e) **swap-array-exp**: For swapping elements, we should go to the indexes which are given in the parameters of the array. Using nested let expressions, we extracted elements in the body of expression thanks to the search helper procedure. So, it becomes easy to swap them while knowing their refs and values. To do that, setref! procedure is used as follows: (setref! (+ array-ref index1(2)) element2(1)).

f) **copy-array-exp**: Purpose of this procedure is to deep copy the array input. While having array-val/ref of the array, we can easily reach its length by looking at the previous ref. So, length will be the same and we assigned its value to the new reference. Then now, we can say that the array-val/ref of the new copied array is the next reference of the new reference containing length. copy-array-helper procedure processes deep copy operation like new-array-helper but it gets the values from array input for new references. It goes over the references and creates a copy by scanning 1 by 1 thanks to the newref procedure after obtaining value on the ref by using deref. Finally, this expression returns the array-val of the copy array.

Part B)

Queue Implementation:

a) newqueue-exp: As advised, we used our existing array implementation to implement queue functionalities. It was stated that the maximum number of enqueue operations would be 1000. According to that, we created an array with a length of 1003. As the first element of the array, we store the index of the top element of the queue. As the second element of the array, we store the index of the tail + 1(or endpoint + 1) of the queue. The other elements are reserved for the queue elements. That is the reason behind the length of the array. As we initialize the queue, We set the top element index and end element index to the same value (which correspond to the third element of the array). This also signifies that the queue is empty.

b) enqueue-exp: We get the index of the queue's endpoint from the relevant part of the array. We put the new enqueued value to that index and increment the endpoint index.

c) dequeue-exp: First, we check whether top-point and end-point indices are equal. If they are equal, it means the queue is empty and dequeue returns -1. If the indices are not equal, we fetch the value stored at the top-point index of the array and return it. Before we return it, we increment the top-point index by one.

d) queue-size-exp: Our implementation of top-point and end-point indices make it very easy to find the size of the queue. The difference between end-index and top-index gives us the queue size.

e) peek-exp: We dereference the value at the top-index position of the array and return it. We do not do any kind of addition or deletion.

f) empty-queue-exp: Similar to queue size query, we check the top-index and end-index. If they are equal, it means the queue is empty and we return true. If not, we return false.

g) print-queue-exp: We get the reference to the array, top-index, end-index and give them to a helper procedure. Helper procedure starts from the top-index and prints the queue elements iteratively. It stops when it reaches the end-index.

Part C) Bonus

map-exp: We got inspired from the existing proc-exp and call-exp implementations. We used a helper function that gets the array reference, array length and the procedure to apply on the array elements. Helper starts from the beginning of the array and iterates through the array while taking the array elements as inputs and replacing the elements with their procedure outputs.

* Our code passes all tests and we believe that all of its parts work properly.

Workload Distribution

We did the syntax specifications part (lang.scm) and array data structure specification part (data-structures.scm) together. We also implemented the interpretation of the newarray() procedure together. Murat did the other array functionalities and print-queue() procedure. Özkan implemented the rest of array-based queue features and functionalities. We did the bonus map implementation together. We gave each other some ideas regarding implementations during the whole project.